

Database Manager



Apr 07, 2024 13:09

Return Types

SQL_ACTION_TYPES JSColumn JSDataSet JSFoundSetUpdater JSRecordMarker JSRecordMarkers JSRecord JSFoundSet JSTable QBSelect QBColumn QBCase QBCaseWhen QBColumn QBColumns QBCondition QBColumn QBGroupBy QBJoin QBJoins QBLogicalCondition QBLogicalCondition QBResult QBColumn QBSort QBSorts QBTableClause QBPart QBParameter QBParameters QBFunctions QUERY_COLUMN_TYPES JSFoundSet JSRecord

Supported Clients

SmartClient WebClient NGClient MobileClient

Property Summary

Boolean	nullColumnValidatorEnabled	Enable/disable the default null validator for non null columns, makes it possible todo the checks later on when saving, when for example autosave is disabled.
---------	----------------------------	--

Methods Summary

Boolean	acquireLock(foundset, recordIndex)	Request lock(s) for a foundset, can be a normal or related foundset.
Boolean	acquireLock(foundset, recordIndex, lockName)	Request lock(s) for a foundset, can be a normal or related foundset.
Boolean	addTableFilterParam(query)	Adds a filter based on a query to all the foundsets based on a table.
Boolean	addTableFilterParam(query, filterName)	Adds a filter based on a query to all the foundsets based on a table.
Boolean	addTableFilterParam(datasource, dataprovider, operator, value)	Adds a filter to all the foundsets based on a table.
Boolean	addTableFilterParam(datasource, dataprovider, operator, value, filterName)	Adds a filter to all the foundsets based on a table.
Boolean	addTableFilterParam(serverName, tableName, dataprovider, operator, value)	Adds a filter to all the foundsets based on a table.
Boolean	addTableFilterParam(serverName, tableName, dataprovider, operator, value, filterName)	Adds a filter to all the foundsets based on a table.
void	addTrackingInfo(columnName, value)	Add tracking info used in the log table.
Boolean	commitTransaction()	Returns true if a transaction is committed; rollback if commit fails.
Boolean	commitTransaction(saveFirst)	Returns true if a transaction is committed; rollback if commit fails.
Boolean	commitTransaction(saveFirst, revertSavedRecords)	Returns true if a transaction is committed; rollback if commit fails.
JSFoundSet	convertFoundSet(foundset, related)	Creates a foundset that combines all the records of the specified one-to-many relation seen from the given parent/primary foundset.
JSFoundSet	convertFoundSet(foundset, related)	Creates a foundset that combines all the records of the specified one-to-many relation seen from the given parent/primary foundset.
JSDataSet	convertToDataSet(foundset)	Converts the argument to a JSDataSet, possible use in controller.
JSDataSet	convertToDataSet(foundset, dataproviderNames)	Converts the argument to a JSDataSet, possible use in controller.
JSDataSet	convertToDataSet(values)	Converts the argument to a JSDataSet, possible use in controller.
JSDataSet	convertToDataSet(values, dataproviderNames)	Converts the argument to a JSDataSet, possible use in controller.
JSDataSet	convertToDataSet(ids)	Converts the argument to a JSDataSet, possible use in controller.
Boolean	copyMatchingFields(source, destination)	Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names).
Boolean	copyMatchingFields(source, destination, overwrite)	Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names).
Boolean	copyMatchingFields(source, destination, names)	Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names).
String	createDataSourceByQuery(name, query, useTableFilters, max_returned_rows, types, pkNames)	Performs a query and saves the result in a datasource.
String	createDataSourceByQuery(name, query, max_returned_rows)	Performs a query and saves the result in a datasource.
String	createDataSourceByQuery(name, query, max_returned_rows, types)	Performs a query and saves the result in a datasource.
String	createDataSourceByQuery(name, query, max_returned_rows, types, pkNames)	Performs a query and saves the result in a datasource.
String	createDataSourceByQuery(name, server_name, sql_query, arguments, max_returned_rows)	Performs a sql query on the specified server, saves the the result in a datasource.

String	<code>createDataSourceByQuery(name, server_name, sql_query, arguments, max_returned_rows, types)</code>	Performs a sql query on the specified server, saves the the result in a datasource.
String	<code>createDataSourceByQuery(name, server_name, sql_query, arguments, max_returned_rows, columnTypes, pkNames)</code>	Performs a sql query on the specified server, saves the the result in a datasource.
JSDataSet	<code>createEmptyDataSet()</code>	Returns an empty dataset object.
JSDataSet	<code>createEmptyDataSet(rowCount, columnCount)</code>	Returns an empty dataset object.
JSDataSet	<code>createEmptyDataSet(rowCount, columnNames)</code>	Returns an empty dataset object.
QBSelect	<code>createSelect(dataSource)</code>	Create a QueryBuilder object for a datasource.
QBSelect	<code>createSelect(dataSource, tableAlias)</code>	Create a QueryBuilder object for a datasource with given table alias.
Boolean	<code>dataSourceExists(dataSource)</code>	Check wether a data source exists.
void	<code>flushCalculations(datasource, onlyUnstored)</code>	This method differences for recalculate() that it only works on a datasource rows/records that are loaded in memory.
void	<code>flushCalculations(datasource, onlyUnstored, calnames)</code>	This method differences for recalculate() that it only works on a datasource rows/records that are loaded in memory.
Boolean	<code>getAutoSave()</code>	Returns true or false if autosave is enabled or disabled.
Array	<code>getDataModelClonesFrom(serverName)</code>	Retrieves a list with names of all database servers that have property DataModelCloneFrom equal to the server name parameter.
JSDataSet	<code>getDataSetByQuery(query, useTableFilters, max_returned_rows)</code>	Performs a sql query with a query builder object.
JSDataSet	<code>getDataSetByQuery(query, max_returned_rows)</code>	Performs a sql query with a query builder object.
JSDataSet	<code>getDataSetByQuery(server_name, sql_query, arguments, max_returned_rows)</code>	Performs a sql query on the specified server, returns the result in a dataset.
String	<code>getDataSource(serverName, tableName)</code>	Returns the datasource corresponding to the given server/table.
String	<code>getDataSourceServerName(dataSource)</code>	Returns the server name from the datasource, or null if not a database datasource.
String	<code>getDataSourceTableName(dataSource)</code>	Returns the table name from the datasource, or null if not a database datasource.
String	<code>getDatabaseProductName(serverName)</code>	Returns the database product name as supplied by the driver for a server.
Array	<code>getEditedRecords()</code>	Returns an array of edited records with outstanding (unsaved) data.
Array	<code>getEditedRecords(foundset)</code>	Returns an array of edited records with outstanding (unsaved) data.
Array	<code>getEditedRecords(datasource)</code>	Returns an array of edited records with outstanding (unsaved) data.
Array	<code>getEditedRecords(datasource, filter)</code>	Returns an array of edited records with outstanding (unsaved) data for a datasource with a filter.
Array	<code>getFailedRecords()</code>	Returns an array of records that fail after a save.
Array	<code>getFailedRecords(foundset)</code>	Returns an array of records that fail after a save.
JSFoundSet	<code>getFoundSet(query)</code>	Returns a foundset object for a specified pk query.
JSFoundSet	<code>getFoundSet(dataSource)</code>	Returns a foundset object for a specified datasource or server and tablename.
JSFoundSet	<code>getFoundSet(serverName, tableName)</code>	Returns a foundset object for a specified datasource or server and tablename.
Number	<code>getFoundSetCount(foundset)</code>	Returns the total number of records in a foundset.
JSFoundSetUpdater	<code>getFoundSetUpdater(foundset)</code>	Returns a JSFoundSetUpdater object that can be used to update all or a specific number of rows in the specified foundset.
JSFoundSet	<code>getNamedFoundSet(name)</code>	An existing foundset under that name will be returned, or created if there is a definition (there is a form with a named foundset property with that name).
Object	<code>getNextSequence(dataSource, columnName)</code>	Gets the next sequence for a column which has a sequence defined in its column dataprovider properties.
String	<code>getSQL(foundsetOrQBSelect)</code>	Returns the internal SQL which defines the specified (related)foundset.
String	<code>getSQL(foundsetOrQBSelect, includeFilters)</code>	Returns the internal SQL which defines the specified (related)foundset.
Array	<code>getSQLParameters(foundsetOrQBSelect)</code>	Returns the internal SQL parameters, as an array, that are used to define the specified (related)foundset.
Array	<code>getSQLParameters(foundsetOrQBSelect, includeFilters)</code>	Returns the internal SQL parameters, as an array, that are used to define the specified (related)foundset.
Array	<code>getServerNames()</code>	Returns an array with all the server names used in the solution.
JSTable	<code>getTable(foundset)</code>	Returns the JSTable object from which more info can be obtained (like columns).
JSTable	<code>getTable(record)</code>	Returns the JSTable object from which more info can be obtained (like columns).
JSTable	<code>getTable(dataSource)</code>	Returns the JSTable object from which more info can be obtained (like columns).
JSTable	<code>getTable(serverName, tableName)</code>	Returns the JSTable object from which more info can be obtained (like columns).
Number	<code>getTableCount(dataSource)</code>	Returns the total number of records(rows) in a table.
Array	<code>getTableFilterParams(serverName)</code>	Returns a two dimensional array object containing the table filter information currently applied to the servers tables.
Array	<code>getTableFilterParams(serverName, filterName)</code>	Returns a two dimensional array object containing the table filter information currently applied to the servers tables.
Array	<code>getTableNames(serverName)</code>	Returns an array of all table names for a specified server.
JSFoundSet	<code>getViewFoundSet(name)</code>	Returns a ViewFoundSet that was created by getViewFoundSet(name,query,register) with the registerd boolean "true".
JSFoundSet	<code>getViewFoundSet(name, query)</code>	Returns a foundset object for a specified query.
JSFoundSet	<code>getViewFoundSet(name, query, register)</code>	Returns a foundset object for a specified query.
Array	<code>getViewNames(serverName)</code>	Returns an array of all view names for a specified server.
Boolean	<code>hasLocks()</code>	Returns true if the current client has any or the specified lock(s) acquired.
Boolean	<code>hasLocks(lockName)</code>	Returns true if the current client has any or the specified lock(s) acquired.
Boolean	<code>hasNewRecords(foundset)</code>	Returns true if the argument (foundSet / record) has at least one row that was not yet saved in the database.

Boolean	<code>hasNewRecords(foundset, index)</code>	Returns true if the argument (foundSet / record) has at least one row that was not yet saved in the database.
Boolean	<code>hasRecordChanges(foundset)</code>	Returns true if the specified foundset, on a specific index or in any of its records, or the specified record has changes or is new unsaved record.
Boolean	<code>hasRecordChanges(foundset, index)</code>	Returns true if the specified foundset, on a specific index or in any of its records, or the specified record has changes or is new unsaved record.
Boolean	<code>hasRecords(foundset)</code>	Returns true if the (related)foundset exists and has records.
Boolean	<code>hasRecords(record, relationString)</code>	Returns true if the (related)foundset exists and has records.
Boolean	<code>hasTransaction()</code>	Returns true if there is an transaction active for this client.
Boolean	<code>mergeRecords(sourceRecord, combinedDestinationRecord)</code>	Merge records from the same foundset, updates entire datamodel (via foreign type on columns) with destination record pk, deletes source record.
Boolean	<code>mergeRecords(sourceRecord, combinedDestinationRecord, columnNames)</code>	Merge records from the same foundset, updates entire datamodel (via foreign type on columns) with destination record pk, deletes source record.
void	<code>recalculate(foundsetOrRecord)</code>	Can be used to recalculate a specified record or all rows in the specified foundset.
Boolean	<code>refreshRecordFromDatabase(foundset, index)</code>	Flushes the client data cache and requeries the data for a record (based on the record index) in a foundset or all records in the foundset.
Boolean	<code>releaseAllLocks()</code>	Release all current locks the client has (optionally limited to named locks).
Boolean	<code>releaseAllLocks(lockName)</code>	Release all current locks the client has (optionally limited to named locks).
Boolean	<code>removeDataSource(uri)</code>	Free resources allocated for a previously created data source.
Boolean	<code>removeTableFilterParam(serverName, filterName)</code>	Removes a previously defined table filter.
void	<code>revertEditedRecords()</code>	Reverts outstanding (not saved) in memory changes from edited records.
void	<code>revertEditedRecords(foundset)</code>	Reverts outstanding (not saved) in memory changes from edited records.
void	<code>rollbackTransaction()</code>	Rollback a transaction started by databaseManager.
void	<code>rollbackTransaction(rollbackEdited)</code>	Rollback a transaction started by databaseManager.
void	<code>rollbackTransaction(rollbackEdited, revertSavedRecords)</code>	Rollback a transaction started by databaseManager.
Boolean	<code>saveData()</code>	Saves all outstanding (unsaved) data and exits the current record.
Boolean	<code>saveData(foundset)</code>	Saves all outstanding (unsaved) data and exits the current record.
Boolean	<code>saveData(record)</code>	Saves all outstanding (unsaved) data and exits the current record.
Boolean	<code>saveData(records)</code>	Saves all outstanding (unsaved) data and exits the current record.
Boolean	<code>setAutoSave(autoSave)</code>	Set autosave, if false then no saves will happen by the ui (not including deletes!).
void	<code>setCreateEmptyFormFoundsets()</code>	Turnoff the initial form foundset record loading, set this in the solution open method.
void	<code>startTransaction()</code>	Start a database transaction.
Boolean	<code>switchServer(sourceName, destinationName)</code>	Switches a named server to another named server with the same datamodel (recommended to be used in an onOpen method for a solution).
Boolean	<code>updateTableFilterParam(serverName, filterName, query)</code>	Updates a previously defined table filter.
Boolean	<code>updateTableFilterParam(serverName, filterName, tableName, dataprovider, operator, value)</code>	Updates a filter with a new condition.
JSRecordMarkers	<code>validate(record)</code>	Validates the given record, it runs first the method that is attached to the entity event "onValidate".
JSRecordMarkers	<code>validate(record, customObject)</code>	Validates the given record, it runs first the method that is attached to the entity event "onValidate".

Property Details

nullColumnValidatorEnabled

Enable/disable the default null validator for non null columns, makes it possible todo the checks later on when saving, when for example autosave is disabled.

Returns

Boolean

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.nullColumnValidatorEnabled = false;//disable

//test if enabled
if(databaseManager.nullColumnValidatorEnabled) application.output('null validation enabled')
```

Methods Details

acquireLock(foundset, recordIndex)

Request lock(s) for a foundset, can be a normal or related foundset.
 The record_index can be -1 to lock all rows, 0 to lock the current row, or a specific row of > 0
 Optionally name the lock(s) so that it can be referenced it in releaseAllLocks()

By default this call doesn't try to lock records in the database itself. But the locks are tracked in the Servoy Server itself.
 If you need database locking because of others applications that can also read the table or you use the Broadcaster plugin for more then 1 servoy server on the same database, you need to set the property 'servoy.record.lock.lockInDB' in the servoy.properties file to true. This will try to do a 'select for update no wait' on databases that supports this.
 This can only be used together with a transaction, so before you aquire the lock a transaction must be started so the database lock is held on to the transaction connection.

Do not change the record data before that, because aquirelock will make sure with a select from the database that it really has the latest data.
 If there are changes to columns that you changed before calling aquireLock these changes will be reverted, so you don't change something again that you didn't see really the value of first.

returns true if the lock could be acquired.

Parameters

[JSFoundSet](#) foundset The JSFoundset to get the lock for
[Number](#) recordIndex The record index which should be locked.

Returns

[Boolean](#) true if the lock could be acquired.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//locks the complete foundset
databaseManager.acquireLock(foundset,-1);

//locks the current row
databaseManager.acquireLock(foundset,0);

//locks all related orders for the current Customer
var success = databaseManager.acquireLock(Cust_to_Orders,-1);
if(!success)
{
    plugins.dialogs.showWarningDialog('Alert','Failed to get a lock','OK');
}
```

acquireLock(foundset, recordIndex, lockName)

Request lock(s) for a foundset, can be a normal or related foundset.
 The record_index can be -1 to lock all rows, 0 to lock the current row, or a specific row of > 0
 Optionally name the lock(s) so that it can be referenced it in releaseAllLocks()

By default this call doesn't try to lock records in the database itself. But the locks are tracked in the Servoy Server itself.
 If you need database locking because of others applications that can also read the table or you use the Broadcaster plugin for more then 1 servoy server on the same database, you need to set the property 'servoy.record.lock.lockInDB' in the servoy.properties file to true. This will try to do a 'select for update no wait' on databases that supports this.
 This can only be used together with a transaction, so before you aquire the lock a transaction must be started so the database lock is held on to the transaction connection.

Do not change the record data before that, because aquirelock will make sure with a select from the database that it really has the latest data.
 If there are changes to columns that you changed before calling aquireLock these changes will be reverted, so you don't change something again that you didn't see really the value of first.

returns true if the lock could be acquired.

Parameters

[JSFoundSet](#) foundset The JSFoundset to get the lock for
[Number](#) recordIndex The record index which should be locked.
[String](#) lockName The name of the lock.

Returns

[Boolean](#) true if the lock could be acquired.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//locks the complete foundset
databaseManager.acquireLock(foundset,-1);

//locks the current row
databaseManager.acquireLock(foundset,0);

//locks all related orders for the current Customer
var success = databaseManager.acquireLock(Cust_to_Orders,-1);
if(!success)
{
    plugins.dialogs.showWarningDialog('Alert','Failed to get a lock','OK');
}
```

addTableFilterParam(query)

Adds a filter based on a query to all the foundsets based on a table.

Filters on tables touched in the query will not be applied to the query filter.
For example, when a table filter exists on the order_details table,
a query filter with a join from orders to order_details will be applied to queries on the orders table,
but the filter condition on the orders_details table will not be included.

returns true if the tablefilter could be applied.

Parameters

[QBSelect](#) query condition to filter on.

Returns

[Boolean](#) true if the tablefilter could be applied.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// Best way to call this in a global solution startup method, but filters may be added/removed at any time.
// Note that

var query = datasources.db.example_data.orders.createSelect();
query.where.add(
    query.or.add(
        query.columns.shipcity.eq('Amersfoort'))
    .add( query.columns.shipcity.eq('Amsterdam')));

var success = databaseManager.addTableFilterParam(query, 'cityFilter')
```

addTableFilterParam(query, filterName)

Adds a filter based on a query to all the foundsets based on a table.

Filters on tables touched in the query will not be applied to the query filter.
For example, when a table filter exists on the order_details table,
a query filter with a join from orders to order_details will be applied to queries on the orders table,
but the filter condition on the orders_details table will not be included.

returns true if the tablefilter could be applied.

Parameters

[QBSelect](#) query condition to filter on.

[String](#) filterName The specified name of the database table filter.

Returns

[Boolean](#) true if the tablefilter could be applied.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// Best way to call this in a global solution startup method, but filters may be added/removed at any time.
// Note that

var query = datasources.db.example_data.orders.createSelect();
query.where.add(
  query.or.add(
    query.columns.shipcity.eq('Amersfoort'))
  .add( query.columns.shipcity.eq('Amsterdam')));

var success = databaseManager.addTableFilterParam(query, 'cityFilter')
```

addTableFilterParam(datasource, dataprovider, operator, value)

Adds a filter to all the foundsets based on a table.

Note: if null is provided as the tablename the filter will be applied on all tables with the dataprovider name. A dataprovider can have multiple filters defined, they will all be applied.

returns true if the tablefilter could be applied.

Parameters

Stringdatasour The datasource
ce

Stringdatapro A specified dataprovider column name.
vider

Stringoperator One of "=", "<", ">", ">=", "<=", "!=", "LIKE", or "IN" optionally augmented with modifiers "#" (ignore case) or "^|" (or-is-null), prefix with "sql:" to allow the value to be interpreted as a custom query.

Object value The specified filter value.
ect

Returns

Boolean true if the tablefilter could be applied.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// Best way to call this in a global solution startup method, but filters may be added/removed at any time.
// Note that multiple filters can be added to the same dataprovider, they will all be applied.

// filter on messages table where messagesid>10, the filter has a name so it can be removed using
databaseManager.removeTableFilterParam()
var success = databaseManager.addTableFilterParam('admin', 'messages', 'messagesid', '>', 10,
'higNumberedMessagesRule')

// all tables that have the companyid column should be filtered
var success = databaseManager.addTableFilterParam('crm', null, 'companyidid', '=', currentcompanyid)

// some filters with in-conditions
var success = databaseManager.addTableFilterParam('crm', 'products', 'productcode', 'in', [120, 144, 200])
// use "sql:in" in stead of "in" to allow the value to be interpreted as a custom query
var success = databaseManager.addTableFilterParam('crm', 'orders', 'countrycode', 'sql:in', 'select country code
from countries where region = "Europe"')

// you can use modifiers in the operator as well, filter on companies where companyname is null or equals-ignore-
case 'servoy'
var success = databaseManager.addTableFilterParam('crm', 'companies', 'companyname', '#^|!=", 'servoy')

// the value may be null, this will result in 'column is null' sql condition.
var success = databaseManager.addTableFilterParam('crm', 'companies', 'verified', '=', null)

//if you want to add a filter for a column (created by you) in the i18n table
databaseManager.addTableFilterParam('database', 'your_i18n_table', 'message_variant', 'in', [1, 2])
```

addTableFilterParam(datasource, dataprovider, operator, value, filterName)

Adds a filter to all the foundsets based on a table.

Note: if null is provided as the tablename the filter will be applied on all tables with the dataprovider name. A dataprovider can have multiple filters defined, they will all be applied.

returns true if the tablefilter could be applied.

Parameters

Stringdatasour The datasource
ce

Stringdataprovider A specified dataprovider column name.
ider

Stringoperator One of "=", "<", ">=", "<=", "!=", LIKE, or IN" optionally augmented with modifiers "#" (ignore case) or "^|" (or-is-null), prefix with "sql:" to allow the value to be interpreted as a custom query.

Object value The specified filter value.
ect

StringfilterNameThe specified name of the database table filter.

Returns

Boolean true if the tablefilter could be applied.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// Best way to call this in a global solution startup method, but filters may be added/removed at any time.
// Note that multiple filters can be added to the same dataprovider, they will all be applied.

// filter on messages table where messagesid>10, the filter has a name so it can be removed using
databaseManager.removeTableFilterParam()
var success = databaseManager.addTableFilterParam('admin', 'messages', 'messagesid', '>', 10,
'highNumberedMessagesRule')

// all tables that have the companyid column should be filtered
var success = databaseManager.addTableFilterParam('crm', null, 'companyid', '=', currentcompanyid)

// some filters with in-conditions
var success = databaseManager.addTableFilterParam('crm', 'products', 'productcode', 'in', [120, 144, 200])
// use "sql:in" in stead of "in" to allow the value to be interpreted as a custom query
var success = databaseManager.addTableFilterParam('crm', 'orders', 'countrycode', 'sql:in', 'select country code
from countries where region = "Europe"')

// you can use modifiers in the operator as well, filter on companies where companyname is null or equals-ignore-
case 'servoy'
var success = databaseManager.addTableFilterParam('crm', 'companies', 'companyname', '^|=', 'servoy')

// the value may be null, this will result in 'column is null' sql condition.
var success = databaseManager.addTableFilterParam('crm', 'companies', 'verified', '=', null)

//if you want to add a filter for a column (created by you) in the i18n table
databaseManager.addTableFilterParam('database', 'your_i18n_table', 'message_variant', 'in', [1, 2])
```

addTableFilterParam(serverName, tableName, dataprovider, operator, value)

Adds a filter to all the foundsets based on a table.

Note: if null is provided as the tablename the filter will be applied on all tables with the dataprovider name.

A dataprovider can have multiple filters defined, they will all be applied.

returns true if the tablefilter could be applied.

Parameters

StringserverName The name of the database server connection for the specified table name.
me

StringtableName The name of the specified table.
me

Stringdataprovider A specified dataprovider column name.
ider

Stringoperator One of "=", "<", ">=", "<=", "!=", LIKE, or IN" optionally augmented with modifiers "#" (ignore case) or "^|" (or-is-null), prefix with "sql:" to allow the value to be interpreted as a custom query.

Object value The specified filter value.
ect

Returns

Boolean true if the tablefilter could be applied.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// Best way to call this in a global solution startup method, but filters may be added/removed at any time.
// Note that multiple filters can be added to the same dataprovider, they will all be applied.

// filter on messages table where messagesid>10, the filter has a name so it can be removed using
databaseManager.removeTableFilterParam()
var success = databaseManager.addTableFilterParam('admin', 'messages', 'messagesid', '>', 10,
'higNumberedMessagesRule')

// all tables that have the companyid column should be filtered
var success = databaseManager.addTableFilterParam('crm', null, 'companyidid', '=', currentcompanyid)

// some filters with in-conditions
var success = databaseManager.addTableFilterParam('crm', 'products', 'productcode', 'in', [120, 144, 200])
// use "sql:in" in stead of "in" to allow the value to be interpreted as a custom query
var success = databaseManager.addTableFilterParam('crm', 'orders', 'countrycode', 'sql:in', 'select country code
from countries where region = "Europe"')

// you can use modifiers in the operator as well, filter on companies where companyname is null or equals-ignore-
case 'servoy'
var success = databaseManager.addTableFilterParam('crm', 'companies', 'companyname', '#^||=', 'servoy')

// the value may be null, this will result in 'column is null' sql condition.
var success = databaseManager.addTableFilterParam('crm', 'companies', 'verified', '=', null)

//if you want to add a filter for a column (created by you) in the i18n table
databaseManager.addTableFilterParam('database', 'your_i18n_table', 'message_variant', 'in', [1, 2])
```

addTableFilterParam(serverName, tableName, dataprovider, operator, value, filterName)

Adds a filter to all the foundsets based on a table.

Note: if null is provided as the tablename the filter will be applied on all tables with the dataprovider name. A dataprovider can have multiple filters defined, they will all be applied.

returns true if the tablefilter could be applied.

Parameters

String serverName The name of the database server connection for the specified table name.
String tableName The name of the specified table.
String dataprovider A specified dataprovider column name.
String operator One of "=", "<", ">=", "<=", "!=", LIKE, or IN" optionally augmented with modifiers "#" (ignore case) or "^||" (or-is-null), prefix with "sql:" to allow the value to be interpreted as a custom query..
Object value The specified filter value.
String filterName The specified name of the database table filter.

Returns

Boolean true if the tablefilter could be applied.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
// Best way to call this in a global solution startup method, but filters may be added/removed at any time.
// Note that multiple filters can be added to the same dataprovider, they will all be applied.

// filter on messages table where messagesid>10, the filter has a name so it can be removed using
databaseManager.removeTableFilterParam()
var success = databaseManager.addTableFilterParam('admin', 'messages', 'messagesid', '>', 10,
'higNumberedMessagesRule')

// all tables that have the companyid column should be filtered
var success = databaseManager.addTableFilterParam('crm', null, 'companyidid', '=', currentcompanyid)

// some filters with in-conditions
var success = databaseManager.addTableFilterParam('crm', 'products', 'productcode', 'in', [120, 144, 200])
// use "sql:in" in stead of "in" to allow the value to be interpreted as a custom query
var success = databaseManager.addTableFilterParam('crm', 'orders', 'countrycode', 'sql:in', 'select country code
from countries where region = "Europe"')

// you can use modifiers in the operator as well, filter on companies where companyname is null or equals-ignore-
case 'servoy'
var success = databaseManager.addTableFilterParam('crm', 'companies', 'companyname', '#^||=', 'servoy')

// the value may be null, this will result in 'column is null' sql condition.
var success = databaseManager.addTableFilterParam('crm', 'companies', 'verified', '=', null)

//if you want to add a filter for a column (created by you) in the i18n table
databaseManager.addTableFilterParam('database', 'your_i18n_table', 'message_variant', 'in', [1, 2])
```

addTrackingInfo(columnName, value)

Add tracking info used in the log table.
 When tracking is enabled and a new row is inserted in the log table,
 if it has a column named 'columnName', its value will be set with 'value'

Parameters

String columnName The name of the column in the log table, used for tracking info
Object value The value to be set when inserting a new row in the log table, for the 'columnName' column

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
databaseManager.addTrackingInfo('log_column_name', 'trackingInfo')
```

commitTransaction()

Returns true if a transaction is committed; rollback if commit fails.
 Saves all edited records and commits the data.

Returns

Boolean if the transaction could be committed.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceled or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
    databaseManager.rollbackTransaction();
}
```

commitTransaction(saveFirst)

Returns true if a transaction is committed; rollback if commit fails.

Parameters

[Boolean](#) saveFirst save edited records to the database first (default true)

Returns

[Boolean](#) if the transaction could be committed.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceled or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
    databaseManager.rollbackTransaction();
}
```

commitTransaction(saveFirst, revertSavedRecords)

Returns true if a transaction is committed; rollback if commit fails.

Parameters

[Boolean](#) saveFirst save edited records to the database first (default true)

[Boolean](#) revertSavedRecords if a commit fails and a rollback is done, the when given false the records are not reverted to the database state (and are in edited records again)

Returns

[Boolean](#) if the transaction could be committed.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceled or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
    databaseManager.rollbackTransaction();
}
```

convertFoundSet(foundset, related)

Creates a foundset that combines all the records of the specified one-to-many relation seen from the given parent/primary foundset. The created foundset will not contain records that have not been saved in the database, because the records in the foundset will be the result of a select query to the database.

Parameters

[JSFoundSet](#) foundset The JSFoundset to convert.

[JSFoundSet](#) related can be a one-to-many relation object or the name of a one-to-many relation

Returns

[JSFoundSet](#) The converted JSFoundset.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// Convert in the order form a orders foundset into a orderdetails foundset,
// that has all the orderdetails from all the orders in the foundset.
var convertedFoundSet = databaseManager.convertFoundSet(foundset,order_to_orderdetails);
// or var convertedFoundSet = databaseManager.convertFoundSet(foundset,"order_to_orderdetails");
forms.orderdetails.controller.showRecords(convertedFoundSet);
```

convertFoundSet(foundset, related)

Creates a foundset that combines all the records of the specified one-to-many relation seen from the given parent/primary foundset.
The created foundset will not contain records that have not been saved in the database, because the records in the foundset will be the result of a select query to the database.

Parameters

[JSFoundSet](#) foundset The JSFoundset to convert.
[String](#) related the name of a one-to-many relation

Returns

[JSFoundSet](#) The converted JSFoundset.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// Convert in the order form a orders foundset into a orderdetails foundset,
// that has all the orderdetails from all the orders in the foundset.
var convertedFoundSet = databaseManager.convertFoundSet(foundset,order_to_orderdetails);
// or var convertedFoundSet = databaseManager.convertFoundSet(foundset,"order_to_orderdetails");
forms.orderdetails.controller.showRecords(convertedFoundSet);
```

convertToDataSet(foundset)

Converts the argument to a JSDataSet, possible use in controller.loadRecords(dataset).
The optional array of dataprovider names is used (only) to add the specified dataprovider names as columns to the dataset.

Parameters

[JSFoundSet](#) foundset The foundset to be converted.

Returns

[JSDataSet](#) JSDataSet with the data.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// converts a foundset pks to a dataset
var dataset = databaseManager.convertToDataSet(foundset);
// converts a foundset to a dataset
//var dataset = databaseManager.convertToDataSet(foundset,['product_id','product_name']);
// converts an object array to a dataset
//var dataset = databaseManager.convertToDataSet(files,['name','path']);
// converts an array to a dataset
//var dataset = databaseManager.convertToDataSet(new Array(1,2,3,4,5,6));
// converts an string list to a dataset
//var dataset = databaseManager.convertToDataSet('4,5,6');
```

convertToDataSet(foundset, dataproviderNames)

Converts the argument to a JSDataSet, possible use in controller.loadRecords(dataset).
The optional array of dataprovider names is used (only) to add the specified dataprovider names as columns to the dataset.

Parameters

[JSFoundSet](#) foundset The foundset to be converted.
[Array](#) dataproviderNames Array with column names.

Returns

[JSDataset](#) JSDataset with the data.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// converts a foundset pks to a dataset
var dataset = databaseManager.convertToDataSet(foundset);
// converts a foundset to a dataset
//var dataset = databaseManager.convertToDataSet(foundset,['product_id','product_name']);
// converts an object array to a dataset
//var dataset = databaseManager.convertToDataSet(files,['name','path']);
// converts an array to a dataset
//var dataset = databaseManager.convertToDataSet(new Array(1,2,3,4,5,6));
// converts a string list to a dataset
//var dataset = databaseManager.convertToDataSet('4,5,6');
```

convertToDataSet(values)

Converts the argument to a JSDataset, possible use in controller.loadRecords(dataset).
The optional array of dataprovider names is used (only) to add the specified dataprovider names as columns to the dataset.

Parameters

[Array](#) values The values array.

Returns

[JSDataset](#) JSDataset with the data.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// converts a foundset pks to a dataset
var dataset = databaseManager.convertToDataSet(foundset);
// converts a foundset to a dataset
//var dataset = databaseManager.convertToDataSet(foundset,['product_id','product_name']);
// converts an object array to a dataset
//var dataset = databaseManager.convertToDataSet(files,['name','path']);
// converts an array to a dataset
//var dataset = databaseManager.convertToDataSet(new Array(1,2,3,4,5,6));
// converts a string list to a dataset
//var dataset = databaseManager.convertToDataSet('4,5,6');
```

convertToDataSet(values, dataproviderNames)

Converts the argument to a JSDataset, possible use in controller.loadRecords(dataset).
The optional array of dataprovider names is used (only) to add the specified dataprovider names as columns to the dataset.

Parameters

[Array](#) values The values array.

[Array](#) dataproviderNames The property names array.

Returns

[JSDataset](#) JSDataset with the data.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// converts a foundset pks to a dataset
var dataset = databaseManager.convertToDataSet(foundset);
// converts a foundset to a dataset
//var dataset = databaseManager.convertToDataSet(foundset,['product_id','product_name']);
// converts an object array to a dataset
//var dataset = databaseManager.convertToDataSet(files,['name','path']);
// converts an array to a dataset
//var dataset = databaseManager.convertToDataSet(new Array(1,2,3,4,5,6));
// converts an string list to a dataset
//var dataset = databaseManager.convertToDataSet('4,5,6');
```

convertToDataSet(ids)

Converts the argument to a JSDataSet, possible use in controller.loadRecords(dataset).
The optional array of dataprovider names is used (only) to add the specified dataprovider names as columns to the dataset.

Parameters

[String](#) ids Concatenated values to be put into dataset.

Returns

[JSDataSet](#) JSDataSet with the data.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// converts a foundset pks to a dataset
var dataset = databaseManager.convertToDataSet(foundset);
// converts a foundset to a dataset
//var dataset = databaseManager.convertToDataSet(foundset,['product_id','product_name']);
// converts an object array to a dataset
//var dataset = databaseManager.convertToDataSet(files,['name','path']);
// converts an array to a dataset
//var dataset = databaseManager.convertToDataSet(new Array(1,2,3,4,5,6));
// converts an string list to a dataset
//var dataset = databaseManager.convertToDataSet('4,5,6');
```

copyMatchingFields(source, destination)

Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names).
The matching requires the properties and getter functions of the source to match those of the destination; for the getter functions, the 'get' will be removed and the remaining name will be converted to lowercase before attempting to match.
Returns true if no error occurred.

NOTE: This function could be used to store a copy of records in an archive table. Use the getRecord() function to get the record as an object.

Before trying this example, please make sure that the foundsets have some records loaded:

Parameters

[Object](#) source The source record or (java/javascript)object to be copied.

[JSRecord](#) destination The destination record to copy to.

Returns

[Boolean](#) true if no errors happened.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

otherfoundset.loadAllRecords();
for( var i = 1 ; i <= foundset.getSize() ; i++ )
{
    var srcRecord = foundset.getRecord(i);
    var destRecord = otherfoundset.getRecord(i);
    if (srcRecord == null || destRecord == null) break;
    databaseManager.copyMatchingFields(srcRecord,destRecord,true)
}
//saves any outstanding changes to the dest foundset
databaseManager.saveData();

//copying from a MailMessage JavaScript object
//var _msg = plugins.mail.receiveMail(login, password, true, 0, null, properties);
//if (_msg != null)
//{
//    controller.newRecord();
//    var srcObject = _msg[0];
//    var destRecord = foundset.getSelectedRecord();
//    databaseManager.copyMatchingFields(srcObject, destRecord, true);
//    databaseManager.saveData();
//}

```

copyMatchingFields(source, destination, overwrite)

Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names).
 The matching requires the properties and getter functions of the source to match those of the destination; for the getter functions, the 'get' will be removed and the remaining name will be converted to lowercase before attempting to match.
 Returns true if no error occurred.

NOTE: This function could be used to store a copy of records in an archive table. Use the getRecord() function to get the record as an object.

Before trying this example, please make sure that the foundsets have some records loaded:

Parameters

Object source The source record or (java/javascript)object to be copied.
JSRecord destination The destination record to copy to.
Boolean overwrite Boolean values to overwrite all values. If overwrite is false/not provided, then the non empty values are not overwritten in the destination record.

Returns

Boolean true if no errors happened.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

otherfoundset.loadAllRecords();
for( var i = 1 ; i <= foundset.getSize() ; i++ )
{
    var srcRecord = foundset.getRecord(i);
    var destRecord = otherfoundset.getRecord(i);
    if (srcRecord == null || destRecord == null) break;
    databaseManager.copyMatchingFields(srcRecord,destRecord,true)
}
//saves any outstanding changes to the dest foundset
databaseManager.saveData();

//copying from a MailMessage JavaScript object
//var _msg = plugins.mail.receiveMail(login, password, true, 0, null, properties);
//if (_msg != null)
//{
//    controller.newRecord();
//    var srcObject = _msg[0];
//    var destRecord = foundset.getSelectedRecord();
//    databaseManager.copyMatchingFields(srcObject, destRecord, true);
//    databaseManager.saveData();
//}

```

copyMatchingFields(source, destination, names)

Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names).
 The matching requires the properties and getter functions of the source to match those of the destination; for the getter functions, the 'get' will be removed and the remaining name will be converted to lowercase before attempting to match.
 Returns true if no error occurred.

NOTE: This function could be used to store a copy of records in an archive table. Use the getRecord() function to get the record as an object.
 Before trying this example, please make sure that the foundsets have some records loaded:

Parameters

Object source The source record or (java/javascript)object to be copied.
JSRecord destination The destination record to copy to.
Array names The property names that shouldn't be overridden.

Returns

Boolean true if no errors happened.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
otherfoundset.loadAllRecords();
for( var i = 1 ; i <= foundset.getSize() ; i++ )
{
    var srcRecord = foundset.getRecord(i);
    var destRecord = otherfoundset.getRecord(i);
    if (srcRecord == null || destRecord == null) break;
    databaseManager.copyMatchingFields(srcRecord,destRecord,true)
}
//saves any outstanding changes to the dest foundset
databaseManager.saveData();

//copying from a MailMessage JavaScript object
//var _msg = plugins.mail.receiveMail(login, password, true, 0, null, properties);
//if (_msg != null)
//{
//    controller.newRecord();
//    var srcObject = _msg[0];
//    var destRecord = foundset.getSelectedRecord();
//    databaseManager.copyMatchingFields(srcObject, destRecord, true);
//    databaseManager.saveData();
//}
```

createDataSourceByQuery(name, query, useTableFilters, max_returned_rows, types, pkNames)

Performs a query and saves the result in a datasource.
 Will throw an exception if anything went wrong when executing the query.
 Column types in the datasource are inferred from the query result or can be explicitly specified.

A datasource can be reused if the data has the same signature (column names and types).
 A new createDataSourceByQuery() call will clear the datasource contents from a previous call and insert the current data.

Parameters

String name Data source name
QBSelect query The query builder to be executed.
Boolean useTableFilters use table filters (default true).
Number max_returned_rows The maximum number of rows returned by the query.
Array types The column types, when null the types are inferred from the query.
Array pkNames array of pk names, when null a hidden pk-column will be added

Returns

String datasource containing the results of the query or null if the parameters are wrong.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// select customer data for order 1234
var q = datasources.db.example_data.customers.createSelect()
q.result.add(q.columns.customer_id).add(q.columns.city).add(q.columns.country);
q.where.add(q.joins.customers_to_orders.columns.orderid.eq(1234));
var uri = databaseManager.createDataSourceByQuery('mydata', q, true, 999, null, ['customer_id']);
//var uri = databaseManager.createDataSourceByQuery('mydata', q, true, 999, [JSColumn.TEXT, JSColumn.TEXT,
JSColumn.TEXT], ['customer_id']);

// the uri can be used to create a form using solution model
var myForm = solutionModel.newForm('newForm', uri, 'myStyleName', false, 800, 600);
myForm.newTextField('city', 140, 20, 140,20);

// the uri can be used to acces a foundset directly
var fs = databaseManager.getFoundSet(uri);
fs.loadAllRecords();
```

createDataSourceByQuery(name, query, max_returned_rows)

Performs a query and saves the result in a datasource.
 Will throw an exception if anything went wrong when executing the query.
 Column types in the datasource are inferred from the query result or can be explicitly specified.

A datasource can be reused if the data has the same signature (column names and types).
 A new createDataSourceByQuery() call will clear the datasource contents from a previous call and insert the current data.

Parameters

String	name	data source name
QBSelect	query	The query builder to be executed.
Number	max_returned_rows	The maximum number of rows returned by the query.

Returns

String datasource containing the results of the query or null if the parameters are wrong.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// select customer data for order 1234
var q = datasources.db.example_data.customers.createSelect()
q.result.add(q.columns.customer_id).add(q.columns.city).add(q.columns.country);
q.where.add(q.joins.customers_to_orders.columns.orderid.eq(1234));
var uri = databaseManager.createDataSourceByQuery('mydata', q, true, 999, null, ['customer_id']);
//var uri = databaseManager.createDataSourceByQuery('mydata', q, true, 999, [JSColumn.TEXT, JSColumn.TEXT,
JSColumn.TEXT], ['customer_id']);

// the uri can be used to create a form using solution model
var myForm = solutionModel.newForm('newForm', uri, 'myStyleName', false, 800, 600);
myForm.newTextField('city', 140, 20, 140,20);

// the uri can be used to acces a foundset directly
var fs = databaseManager.getFoundSet(uri);
fs.loadAllRecords();
```

createDataSourceByQuery(name, query, max_returned_rows, types)

Performs a query and saves the result in a datasource.
 Will throw an exception if anything went wrong when executing the query.
 Column types in the datasource are inferred from the query result or can be explicitly specified.

Using this variation of createDataSourceByQuery any Tablefilter on the involved tables will be taken into account.

A datasource can be reused if the data has the same signature (column names and types).
 A new createDataSourceByQuery() call will clear the datasource contents from a previous call and insert the current data.

Parameters

<code>String</code>	<code>name</code>	Data source name
<code>QBSelect</code>	<code>query</code>	The query builder to be executed.
<code>Number</code>	<code>max_returned_rows</code>	The maximum number of rows returned by the query.
<code>Array</code>	<code>types</code>	The column types

Returns

`String` datasource containing the results of the query or null if the parameters are wrong.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// select customer data for order 1234
var q = datasources.db.example_data.customers.createSelect();
q.result.add(q.columns.address).add(q.columns.city).add(q.columns.country);
q.where.add(q.joins.customers_to_orders.columns.orderid.eq(1234));
var uri = databaseManager.createDataSourceByQuery('mydata', q, 999);
//var uri = databaseManager.createDataSourceByQuery('mydata', q, 999, [JSColumn.TEXT, JSColumn.TEXT, JSColumn.TEXT]);

// the uri can be used to create a form using solution model
var myForm = solutionModel.newForm('newForm', uri, 'myStyleName', false, 800, 600);
myForm.newTextField('city', 140, 20, 140,20);

// the uri can be used to acces a foundset directly
var fs = databaseManager.getFoundSet(uri);
fs.loadAllRecords();
```

createDataSourceByQuery(name, query, max_returned_rows, types, pkNames)

Performs a query and saves the result in a datasource.
Will throw an exception if anything went wrong when executing the query.
Column types in the datasource are inferred from the query result or can be explicitly specified.

Using this variation of createDataSourceByQuery any Tablefilter on the involved tables will be taken into account.

A datasource can be reused if the data has the same signature (column names and types).
A new createDataSourceByQuery() call will clear the datasource contents from a previous call and insert the current data.

Parameters

<code>String</code>	<code>name</code>	Data source name
<code>QBSelect</code>	<code>query</code>	The query builder to be executed.
<code>Number</code>	<code>max_returned_rows</code>	The maximum number of rows returned by the query.
<code>Array</code>	<code>types</code>	The column types
<code>Array</code>	<code>pkNames</code>	array of pk names, when null a hidden pk-column will be added

Returns

`String` datasource containing the results of the query or null if the parameters are wrong.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// select customer data for order 1234
var q = datasources.db.example_data.customers.createSelect();
q.result.add(q.columns.customer_id).add(q.columns.city).add(q.columns.country);
q.where.add(q.joins.customers_to_orders.columns.orderid.eq(1234));
var uri = databaseManager.createDataSourceByQuery('mydata', q, 999, null, ['customer_id']);
//var uri = databaseManager.createDataSourceByQuery('mydata', q, 999, [JSColumn.TEXT, JSColumn.TEXT, JSColumn.TEXT], ['customer_id']);

// the uri can be used to create a form using solution model
var myForm = solutionModel.newForm('newForm', uri, 'myStyleName', false, 800, 600);
myForm.newTextField('city', 140, 20, 140,20);

// the uri can be used to acces a foundset directly
var fs = databaseManager.getFoundSet(uri);
fs.loadAllRecords();
```

createDataSourceByQuery(name, server_name, sql_query, arguments, max_returned_rows)

Performs a sql query on the specified server, saves the the result in a datasource.
 Will throw an exception if anything went wrong when executing the query.
 Column types in the datasource are inferred from the query result or can be explicitly specified.

Using this variation of createDataSourceByQuery any Tablefilter on the involved tables will be disregarded.

A datasource can be reused if the data has the same signature (column names and types).
 A new createDataSourceByQuery() call will clear the datasource contents from a previous call and insert the current data.

Parameters

String	name	data source name
String	server_name	The name of the server where the query should be executed.
String	sql_query	The custom sql, must start with 'select', 'call', 'with' or 'declare'.
Array	arguments	Specified arguments or null if there are no arguments.
Number	max_returned_rows	The maximum number of rows returned by the query.

Returns

String datasource containing the results of the query or null if the parameters are wrong.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var query = 'select address, city, country from customers';
var uri = databaseManager.createDataSourceByQuery('mydata', 'example_data', query, null, 999);
//var uri = databaseManager.createDataSourceByQuery('mydata', 'example_data', query, null, 999, [JSColumn.TEXT,
JSColumn.TEXT, JSColumn.TEXT]);

// the uri can be used to create a form using solution model
var myForm = solutionModel.newForm('newForm', uri, 'myStyleName', false, 800, 600)
myForm.newTextField('city', 140, 20, 140,20)

// the uri can be used to acces a foundset directly
var fs = databaseManager.getFoundSet(uri)
fs.loadAllRecords();
```

createDataSourceByQuery(name, server_name, sql_query, arguments, max_returned_rows, types)

Performs a sql query on the specified server, saves the the result in a datasource.
 Will throw an exception if anything went wrong when executing the query.
 Column types in the datasource are inferred from the query result or can be explicitly specified.

Using this variation of createDataSourceByQuery any Tablefilter on the involved tables will be disregarded.

A datasource can be reused if the data has the same signature (column names and types).
 A new createDataSourceByQuery() call will clear the datasource contents from a previous call and insert the current data.

Parameters

String	name	data source name
String	server_name	The name of the server where the query should be executed.
String	sql_query	The custom sql, must start with 'select', 'call', 'with' or 'declare'.
Array	arguments	Specified arguments or null if there are no arguments.
Number	max_returned_rows	The maximum number of rows returned by the query.
Array	types	The column types

Returns

String datasource containing the results of the query or null if the parameters are wrong.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

var query = 'select address, city, country from customers';
var uri = databaseManager.createDataSourceByQuery('mydata', 'example_data', query, null, 999);
//var uri = databaseManager.createDataSourceByQuery('mydata', 'example_data', query, null, 999, [JSColumn.TEXT,
JSColumn.TEXT, JSColumn.TEXT]);

// the uri can be used to create a form using solution model
var myForm = solutionModel.newForm('newForm', uri, 'myStyleName', false, 800, 600)
myForm.newTextField('city', 140, 20, 140,20)

// the uri can be used to acces a foundset directly
var fs = databaseManager.getFoundSet(uri)
fs.loadAllRecords();

```

createDataSourceByQuery(name, server_name, sql_query, arguments, max_returned_rows, columnTypes, pkNames)

Performs a sql query on the specified server, saves the the result in a datasource.

Will throw an exception if anything went wrong when executing the query.

Column types in the datasource are inferred from the query result or can be explicitly specified.

Using this variation of createDataSourceByQuery any Tablefilter on the involved tables will be disregarded.

A datasource can be reused if the data has the same signature (column names and types).

A new createDataSourceByQuery() call will clear the datasource contents from a previous call and insert the current data.

Parameters

String	name	data source name
String	server_name	The name of the server where the query should be executed.
String	sql_query	The custom sql, must start with 'select', 'call', 'with' or 'declare'.
Array	arguments	Specified arguments or null if there are no arguments.
Number	max_returned_rows	The maximum number of rows returned by the query.
Object	columnTypes	The column types
Array	pkNames	array of pk names, when null a hidden pk-column will be added

Returns

[String](#) datasource containing the results of the query or null if the parameters are wrong.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

var query = 'select customer_id, address, city, country from customers';
var uri = databaseManager.createDataSourceByQuery('mydata', 'example_data', query, null, 999);
//var uri = databaseManager.createDataSourceByQuery('mydata', 'example_data', query, null, 999, [JSColumn.TEXT,
JSColumn.TEXT, JSColumn.TEXT], ['customer_id']);

// the uri can be used to create a form using solution model
var myForm = solutionModel.newForm('newForm', uri, 'myStyleName', false, 800, 600)
myForm.newTextField('city', 140, 20, 140,20)

// the uri can be used to acces a foundset directly
var fs = databaseManager.getFoundSet(uri)
fs.loadAllRecords();

```

createEmptyDataSet()

Returns an empty dataset object.

Returns

[JSDataset](#) An empty JSDataset with the initial sizes.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// gets an empty dataset with a specified row and column count
var dataset = databaseManager.createEmptyDataSet(10,10)
// gets an empty dataset with a specified row count and column array
var dataset2 = databaseManager.createEmptyDataSet(10,new Array ('a','b','c','d'))
```

createEmptyDataSet(rowCount, columnCount)

Returns an empty dataset object.

Parameters

[Number](#) rowCount The number of rows in the DataSet object.
[Number](#) columnCount Number of columns.

Returns

[JSDataSet](#) An empty JSDataSet with the initial sizes.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// gets an empty dataset with a specified row and column count
var dataset = databaseManager.createEmptyDataSet(10,10)
// gets an empty dataset with a specified row count and column array
var dataset2 = databaseManager.createEmptyDataSet(10,new Array ('a','b','c','d'))
```

createEmptyDataSet(rowCount, columnNames)

Returns an empty dataset object.

Parameters

[Number](#) rowCount ;
[Array](#) columnNames;

Returns

[JSDataSet](#) An empty JSDataSet with the initial sizes.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// gets an empty dataset with a specified row and column count
var dataset = databaseManager.createEmptyDataSet(10,10)
// gets an empty dataset with a specified row count and column array
var dataset2 = databaseManager.createEmptyDataSet(10,new Array ('a','b','c','d'))
```

createSelect(dataSource)

Create a QueryBuilder object for a datasource.

Parameters

[String](#) dataSource The data source to build a query for.

Returns

[QBSelect](#) query builder

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
/** @type {QBSelect<db:/example_data/book_nodes>} */
var q = databaseManager.createSelect('db:/example_data/book_nodes');
q.result.addPk()
q.where.add(q.columns.label_text.not.isIn(null))
datasources.db.example_data.book_nodes.getFoundSet().loadRecords(q)
```

createSelect(dataSource, tableAlias)

Create a QueryBuilder object for a datasource with given table alias.
The alias can be used inside custom queries to bind to the outer table.

Parameters

[String](#) dataSource The data source to build a query for.

[String](#) tableAlias The alias for the main table.

Returns

[QBSelect](#) query builder

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
/** @type {QBSelect<db:/example_data/book_nodes>} */
var q = databaseManager.createSelect('db:/example_data/book_nodes', 'b');
q.result.addPk()
q.where.add(q.columns.label_text.isin('select comment_text from book_text t where t.note_text = ? and t.node_id
= b.node_id', ['test']))
datasources.db.example_data.book_nodes.getFoundSet().loadRecords(q)
```

dataSourceExists(dataSource)

Check whether a data source exists. This function can be used for any type of data source (db-based, in-memory).

Parameters

[String](#) dataSource the datasource string to check.

Returns

[Boolean](#) boolean exists

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
if (!databaseManager.dataSourceExists(dataSource))
{
    // does not exist
}
```

flushCalculations(datasource, onlyUnstored)

This method differences for recalculate() that it only works on a datasource rows/records that are loaded in memory.

It will not cause extra rows of that datasource to be loaded in memory (except if a calc itself would do that)

if onlyUnstored is true, then only unstored calculations will be flushed. (so also not causing any saves to the database)

Parameters

[String](#) datasource The datasource to flush all calculations of

[Boolean](#) onlyUnstored to only go over the unstore calcs of this datasource

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// flushed all unstored calculations of the foundsets datasource.
databaseManager.flushCalculations(datasource, true);
```

flushCalculations(datasource, onlyUnstored, calcnames)

This method differences for recalculate() that it only works on a datasource rows/records that are loaded in memory.

It will not cause extra rows of that datasource to be loaded in memory (except if a calc itself would do that)

if onlyUnstored is true, then only unstored calculations will be flushed. (so also not causing any saves to the database)

Parameters

String datasource The datasource to flush all calculations of
Boolean onlyUnstored to only go over the unstore cals of this datasource
Array calnames A string array of calculation names that need to be flushed, if null then all unstored (or all depending on the boolean)

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// flushed all unstored caclulations of the foundsets datasource.
databaseManager.flushCalculations(datasource, true);
```

getAutoSave()

Returns true or false if autosave is enabled or disabled.

Returns

Boolean true if autosave if enabled.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//Set autosave, if false then no saves will happen by the ui (not including deletes!). Until you call saveData
or setAutoSave(true)
//Rollbacks in mem the records that were edited and not yet saved. Best used in combination with autosave false.
databaseManager.setAutoSave(false)
//Now let users input data

//On save or cancel, when data has been entered:
if (cancel) databaseManager.rollbackEditedRecords()
databaseManager.setAutoSave(true)
```

getDataModelClonesFrom(serverName)

Retrieves a list with names of all database servers that have property DataModelCloneFrom equal to the server name parameter.

Parameters

String serverName ;

Returns

Array

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var serverNames = databaseManager.getDataModelClonesFrom('myServerName');
```

getDataSetByQuery(query, useTableFilters, max_returned_rows)

Performs a sql query with a query builder object.
 Will throw an exception if anything did go wrong when executing the query.

Parameters

QBSelect query QBSelect query.
Boolean useTableFilters use table filters (default true).
Number max_returned_rows The maximum number of rows returned by the query.

Returns

JSDataSet The JSDataSet containing the results of the query.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// use the query from a foundset and add a condition
/** @type {QBSelect<db:/example_data/orders>} */
var q = foundset.getQuery()
q.where.add(q.joins.orders_to_order_details.columns.discount.eq(2))
var maxReturnedRows = 10; //useful to limit number of rows
var ds = databaseManager.getDataSetByQuery(q, true, maxReturnedRows);

// query: select PK from example.book_nodes where parent = 111 and(note_date is null or note_date > now)
var query = datasources.db.example_data.book_nodes.createSelect().result.addPk().root
query.where.add(query.columns.parent_id.eq(111))
    .add(query.or
        .add(query.columns.note_date.isNull)
        .add(query.columns.note_date.gt(new Date())))
databaseManager.getDataSetByQuery(q, true, max_returned_rows)
```

getDataSetByQuery(query, max_returned_rows)

Performs a sql query with a query builder object.
Will throw an exception if anything did go wrong when executing the query.

Using this variation of getDataSetByQuery any Tablefilter on the involved tables will be taken into account.

Parameters

[QBSelect](#) query QBSelect query.
[Number](#) max_returned_rows The maximum number of rows returned by the query.

Returns

[JSDataset](#) The JSDataset containing the results of the query.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
// use the query froma foundset and add a condition
/** @type {QBSelect<db:/example_data/orders>} */
var q = foundset.getQuery()
q.where.add(q.joins.orders_to_order_details.columns.discount.eq(2))
var maxReturnedRows = 10; //useful to limit number of rows
var ds = databaseManager.getDataSetByQuery(q, maxReturnedRows);

// query: select PK from example.book_nodes where parent = 111 and(note_date is null or note_date > now)
var query = datasources.db.example_data.book_nodes.createSelect().result.addPk().root
query.where.add(query.columns.parent_id.eq(111))
    .add(query.or
        .add(query.columns.note_date.isNull)
        .add(query.columns.note_date.gt(new Date())))
databaseManager.getDataSetByQuery(q, max_returned_rows)
```

getDataSetByQuery(server_name, sql_query, arguments, max_returned_rows)

Performs a sql query on the specified server, returns the result in a dataset.
Will throw an exception if query is not a select statement or anything did go wrong when executing the query.

Using this variation of getDataSetByQuery any Tablefilter on the involved tables will be disregarded.

Parameters

[String](#) server_name The name of the server where the query should be executed.
[String](#) sql_query The custom sql, must start with 'select', 'call', 'with' or 'declare'.
[Array](#) arguments Specified arguments or null if there are no arguments.
[Number](#) max_returned_rows The maximum number of rows returned by the query.

Returns

[JSDataset](#) The JSDataset containing the results of the query.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
//finds duplicate records in a specified foundset
var vQuery = " SELECT companiesid from companies where company_name IN (SELECT company_name from companies group
bycompany_name having count(company_name)>1 )";
var vDataset = databaseManager.getDataSetByQuery(databaseManager.getDataSourceServerName(controller.
getDataSource()), vQuery, null, 1000);
controller.loadRecords(vDataset);

var maxReturnedRows = 10;//useful to limit number of rows
var query = 'select c1,c2,c3 from test_table where start_date = ?';//do not use '.' or special chars in names or
aliases if you want to access data by name
var args = new Array();
args[0] = order_date //or new Date()
var dataset = databaseManager.getDataSetByQuery(databaseManager.getDataSourceServerName(controller.
getDataSource()), query, args, maxReturnedRows);

// place in label:
// elements.myLabel.text = '<html>'+dataset.getAsHTML()+'</html>';

//example to calc a strange total
global_total = 0;
for( var i = 1 ; i <= dataset.getMaxRowIndex() ; i++ )
{
    dataset.rowIndex = i;
    global_total = global_total + dataset.c1 + dataset.getValue(i,3);
}
//example to assign to dataprovider
//employee_salary = dataset.getValue(row,column)
```

getDataSource(serverName, tableName)

Returns the datasource corresponding to the given server/table.

Parameters

[String](#) serverName The name of the table's server.
[String](#) tableName The table's name.

Returns

[String](#) The datasource of the given table/server.

Supported Clients

SmartClient, WebClient, NGClient, MobileClient

Sample

```
var datasource = databaseManager.getDataSource('example_data', 'categories');
```

getDataSourceServerName(dataSource)

Returns the server name from the datasource, or null if not a database datasource.

Parameters

[String](#) dataSource The datasource string to get the server name from.

Returns

[String](#) The servername of the datasource.

Supported Clients

SmartClient, WebClient, NGClient, MobileClient

Sample

```
var servername = databaseManager.getDataSourceServerName(datasource);
```

getDataSourceTableName(dataSource)

Returns the table name from the datasource, or null if not a database datasource.

Parameters

[String](#) dataSource The datasource string to get the tablename from.

Returns

[String](#) The tablename of the datasource.

Supported Clients

SmartClient,WebClient,NGClient,MobileClient

Sample

```
var tablename = databaseManager.getDataSourceTableName(datasource);
```

getDatabaseProductName(serverName)

Returns the database product name as supplied by the driver for a server.

NOTE: For more detail on named server connections, see the chapter on Database Connections, beginning with the Introduction to database connections in the Servoy Developer User's Guide.

Parameters

[String](#) serverName The specified name of the database server connection.

Returns

[String](#) A database product name.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var databaseProductName = databaseManager.getDatabaseProductName(servername)
```

getEditedRecords()

Returns an array of edited records with outstanding (unsaved) data.

This is different from JSRecord.isEditing() because this call actually checks if there are changes between the current record data and the stored data in the database. If there are no changes then the record is removed from the edited records list (so after this call JSRecord.isEditing() can return false when it returned true just before this call)

NOTE: To return a dataset of outstanding (unsaved) edited data for each record, see JSRecord.getChangedData();
NOTE2: The fields focus may be lost in user interface in order to determine the edits.

Returns

[Array](#) Array of outstanding/unsaved JSRecords.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//This method can be used to loop through all outstanding changes,
//the application.output line contains all the changed data, their tablename and primary key
var editr = databaseManager.getEditedRecords()
for (x=0;x<editr.length;x++)
{
    var ds = editr[x].getChangedData();
    var jstable = databaseManager.getTable(editr[x]);
    var tableSQLName = jstable.getSQLName();
    var pkrec = jstable.getRowIdentifierColumnNames().join(',');
    var pkvals = new Array();
    for (var j = 0; j < jstable.getRowIdentifierColumnNames().length; j++)
    {
        pkvals[j] = editr[x][jstable.getRowIdentifierColumnNames()[j]];
    }
    application.output('Table: '+tableSQLName +', PKs: '+ pkvals.join(',') +' ('+pkrec +')');
    // Get a dataset with outstanding changes on a record
    for( var i = 1 ; i <= ds.getMaxRowIndex() ; i++ )
    {
        application.output('Column: '+ ds.getValue(i,1) +', oldValue: '+ ds.getValue(i,2) +', newValue:
'+ ds.getValue(i,3));
    }
}
//in most cases you will want to set autoSave back on now
databaseManager.setAutoSave(true);
```

getEditedRecords(foundset)

Returns an array of edited records with outstanding (unsaved) data.

NOTE: To return a dataset of outstanding (unsaved) edited data for each record, see JSRecord.getChangedData();

NOTE2: The fields focus may be lost in user interface in order to determine the edits.

Parameters

[JSFoundSet](#) foundset return edited records in the foundset only.

Returns

[Array](#) Array of outstanding/unsaved JSRecords.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//This method can be used to loop through all outstanding changes in a foundset,
//the application.output line contains all the changed data, their tablename and primary key
var editr = databaseManager.getEditedRecords(foundset)
for (x=0;x<editr.length;x++)
{
    var ds = editr[x].getChangedData();
    var jstable = databaseManager.getTable(editr[x]);
    var tableSQLName = jstable.getSQLName();
    var pkrec = jstable.getRowIdentifierColumnNames().join(',');
    var pkvals = new Array();
    for (var j = 0; j < jstable.getRowIdentifierColumnNames().length; j++)
    {
        pkvals[j] = editr[x][jstable.getRowIdentifierColumnNames()[j]];
    }
    application.output('Table: '+tableSQLName +', PKs: '+ pkvals.join(',') +' ('+pkrec +')');
    // Get a dataset with outstanding changes on a record
    for( var i = 1 ; i <= ds.getMaxRowIndex() ; i++ )
    {
        application.output('Column: '+ ds.getValue(i,1) +', oldValue: '+ ds.getValue(i,2) +', newValue:
'+ ds.getValue(i,3));
    }
}
databaseManager.saveData(foundset);//save all records from foundset
```

getEditedRecords(datasource)

Returns an array of edited records with outstanding (unsaved) data.

Parameters

[String](#) datasource the datasource for which to get the edited records

Returns

[Array](#) Array of outstanding/unsaved JSRecords

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// This method can be used to loop through all outstanding changes for a specific datasource.
// The application.output line contains all the changed data, their tablename and primary key
var edits = databaseManager.getEditedRecords(datasources.db.mydb.mytable.getDataSource())

var jsTable = databaseManager.getTable('mydb', 'mytable');
var tableSQLName = jsTable.getSQLName();
var pkColumnNames = jsTable.getRowIdentifierColumnNames().join(',');
var pkValues [];

var x;
var ds;
var i;

for (x = 0; x < edits.length; x++) {
    ds = edits[x].getChangedData();
    pkValues.length = 0;

    for (i = 0; i < jsTable.getRowIdentifierColumnNames().length; i++) {
        pkValues[i] = edits[x][jsTable.getRowIdentifierColumnNames()[i]];
    }

    application.output('Table: ' + tableSQLName + ', PKs: ' + pkValues.join(',') + ' (' + pkColumnNames +
    ')');

    // Output the outstanding changes on each record
    for (i = 1; i <= ds.getMaxRowIndex(); i++) {
        application.output('Column: ' + ds.getValue(i, 1) + ', oldValue: ' + ds.getValue(i, 2) + ',
        newValue: ' + ds.getValue(i, 3));
    }
}
databaseManager.saveData(edits); //save all edited records in the datasource
```

getEditedRecords(datasource, filter)

Returns an array of edited records with outstanding (unsaved) data for a datasource with a filter.

Parameters

[String](#) datasource the datasource for which to get the edited records

[Object](#) filter criteria against which the edited record must match to be included

Returns

[Array](#) Array of outstanding/unsaved JSRecords

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// This method can be used to loop through all outstanding changes for a specific datasource.
// The application.output line contains all the changed data, their tablename and primary key.
// Filter on records that match certain criteria.
// The criteria can be specified in a javascript object, for example get edited records for country NL or DE and
// currency EUR.
var edits = databaseManager.getEditedRecords(datasources.db.mydb.mytable.getDataSource(), {currency: 'EUR',
country: ['NL', 'DE']});

var jsTable = databaseManager.getTable('mydb', 'mytable');
var tableSQLName = jsTable.getSQLName();
var pkColumnNames = jsTable.getRowIdentifierColumnNames().join(',');
var pkValues [];

var x;
var ds;
var i;

for (x = 0; x < edits.length; x++) {
    ds = edits[x].getChangedData();
    pkValues.length = 0;

    for (i = 0; i < jsTable.getRowIdentifierColumnNames().length; i++) {
        pkValues[i] = edits[x][jsTable.getRowIdentifierColumnNames()[i]];
    }

    application.output('Table: ' + tableSQLName + ', PKs: ' + pkValues.join(',') + ' (' + pkColumnNames +
    ')');

    // Output the outstanding changes on each record
    for (i = 1; i <= ds.getMaxRowIndex(); i++) {
        application.output('Column: ' + ds.getValue(i, 1) + ', oldValue: ' + ds.getValue(i, 2) + ',
        newValue: ' + ds.getValue(i, 3));
    }
}
databaseManager.saveData(edits); //save all edited records in the datasource
```

getFailedRecords()

Returns an array of records that fail after a save.

Returns

[Array](#) Array of failed JSRecords

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
var array = databaseManager.getFailedRecords()
for( var i = 0 ; i < array.length ; i++ )
{
    var record = array[i];
    application.output(record.exception);
    if (record.exception.getErrorCode() == ServoyException.RECORD_VALIDATION_FAILED)
    {
        // exception thrown in pre-insert/update/delete event method
        var thrown = record.exception.getValue()
        application.output("Record validation failed: "+thrown)
    }
    // find out the table of the record (similar to getEditedRecords)
    var jstable = databaseManager.getTable(record);
    var tableSQLName = jstable.getSQLName();
    application.output('Table:'+tableSQLName+' in server:'+jstable.getServerName()+ ' failed to save.')
}
```

getFailedRecords(foundset)

Returns an array of records that fail after a save.

Parameters

[JSFoundSet](#) foundset return failed records in the foundset only.

Returns

[Array](#) Array of failed JSRecords

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var array = databaseManager.getFailedRecords(foundset)
for( var i = 0 ; i < array.length ; i++ )
{
    var record = array[i];
    application.output(record.exception);
    if (record.exception.getErrorCode() == ServoyException.RECORD_VALIDATION_FAILED)
    {
        // exception thrown in pre-insert/update/delete event method
        var thrown = record.exception.getValue()
        application.output("Record validation failed: "+thrown)
    }
    // find out the table of the record (similar to getEditedRecords)
    var jstable = databaseManager.getTable(record);
    var tableSQLName = jstable.getSQLName();
    application.output('Table:'+tableSQLName+' in server:'+jstable.getServerName()+' failed to save.')
}
```

getFoundSet(query)

Returns a foundset object for a specified pk query.

Parameters

[QBSelect](#) query The query to get the JSFoundset for.

Returns

[JSFoundSet](#) A new JSFoundset for that query.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// type the foundset returned from the call with JSDoc, fill in the right server/tablename
/** @type {JSFoundSet<db:/servername/tablename>} */
var fs = databaseManager.getFoundSet(controller.getDataSource())
// same as datasources.db.example_data.orders.getFoundSet() or datasources.mem['myds'].getFoundSet()
var ridx = fs.newRecord()
var record = fs.getRecord(ridx)
record.emp_name = 'John'
databaseManager.saveData()
```

getFoundSet(dataSource)

Returns a foundset object for a specified datasource or server and tablename.

Alternative method: `datasources.db.server_name.table_name.getFoundSet()` or `datasources.mem['ds'].getFoundSet()`

Parameters

[String](#) dataSource The datasource to get a JSFoundset for.

Returns

[JSFoundSet](#) A new JSFoundset for that datasource.

Supported Clients

SmartClient,WebClient,NGClient,MobileClient

Sample

```
// type the foundset returned from the call with JSDoc, fill in the right server/tablename
/** @type {JSFoundSet<db:/servername/tablename>} */
var fs = databaseManager.getFoundSet(controller.getDataSource())
// same as datasources.db.example_data.orders.getFoundSet() or datasources.mem['myds'].getFoundSet()
var ridx = fs.newRecord()
var record = fs.getRecord(ridx)
record.emp_name = 'John'
databaseManager.saveData()
```

getFoundSet(serverName, tableName)

Returns a foundset object for a specified datasource or server and tablename.

Parameters

[String](#) serverName The servername to get a JSFoundset for.

[String](#) tableName The tablename for that server

Returns

[JSFoundSet](#) A new JSFoundset for that datasource.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
// type the foundset returned from the call with JSDoc, fill in the right server/tablename
/** @type {JSFoundSet<db:/servername/tablename>} */
var fs = databaseManager.getFoundSet(controller.getDataSource())
// same as datasources.db.example_data.orders.getFoundSet() or datasources.mem['myds'].getFoundSet()
var ridx = fs.newRecord()
var record = fs.getRecord(ridx)
record.emp_name = 'John'
databaseManager.saveData()
```

getFoundSetCount(foundset)

Returns the total number of records in a foundset.

NOTE: This can be an expensive operation (time-wise) if your resultset is large.

Parameters

[JSFoundSet](#) foundset The JSFoundset to get the count for.

Returns

[Number](#) the foundset count

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
//return the total number of records in a foundset.
databaseManager.getFoundSetCount(foundset);
```

getFoundSetUpdater(foundset)

Returns a JSFoundSetUpdater object that can be used to update all or a specific number of rows in the specified foundset.

Parameters

[JSFoundSet](#) foundset The foundset to update.

Returns

[JSFoundSetUpdater](#) The JSFoundSetUpdater for the specified JSFoundset.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
//1) update entire foundset
var fsUpdater = databaseManager.getFoundSetUpdater(foundset)
fsUpdater.setColumn('customer_type',1)
fsUpdater.setColumn('my_flag',0)
fsUpdater.performUpdate()

//2) update part of foundset, for example the first 4 row (starts with selected row)
var fsUpdater = databaseManager.getFoundSetUpdater(foundset)
fsUpdater.setColumn('customer_type',new Array(1,2,3,4))
fsUpdater.setColumn('my_flag',new Array(1,0,1,0))
fsUpdater.performUpdate()

//3) safely loop through foundset (starts with selected row)
controller.setSelectedIndex(1)
var count = 0
var fsUpdater = databaseManager.getFoundSetUpdater(foundset)
while(fsUpdater.next())
{
    fsUpdater.setColumn('my_flag',count++)
}
```

getNamedFoundSet(name)

An existing foundset under that name will be returned, or created if there is a definition (there is a form with a named foundset property with that name).

Alternative method: `datasources.db.server_name.table_name.getFoundSet(name)`

Parameters

[String](#) name The named foundset name

Returns

[JSFoundSet](#) An existing named(separate) foundset.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// type the foundset returned from the call with JSDoc, fill in the right server/tablename
/** @type {JSFoundset<db:/servername/tablename>} */
var fs = databaseManager.getNamedFoundSet('myname')
// same as datasources.db.example_data.orders.getFoundSet('myname')
var ridx = fs.newRecord()
var record = fs.getRecord(ridx)
record.emp_name = 'John'
databaseManager.saveData()
```

getNextSequence(dataSource, columnName)

Gets the next sequence for a column which has a sequence defined in its column dataprovider properties.

NOTE: For more information on configuring the sequence for a column, see the section Auto enter options for a column from the Dataproviders chapter in the Servoy Developer User's Guide.

Parameters

[S](#) data The datasource that points to the table which has the column with the sequence, or the name of the server where the table can be found. If the [tri](#) Sour name of the server is specified, then a second optional parameter specifying the name of the table must be used. If the datasource is specified, [ngce](#) then the name of the table is not needed as the second argument.

[S](#) colu The name of the column that has a sequence defined in its properties.

[tri](#) mnN

[ngame](#)

Returns

[Object](#) The next sequence for the column, null if there was no sequence for that column or if there is no column with the given name.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var seqDataSource = forms.seq_table.controller.getDataSource();
var nextValue = databaseManager.getNextSequence(seqDataSource, 'seq_table_value');
application.output(nextValue);

nextValue = databaseManager.getNextSequence(databaseManager.getDataSourceServerName(seqDataSource),
databaseManager.getDataSourceTableName(seqDataSource), 'seq_table_value')
application.output(nextValue);
```

getSQL(foundsetOrQBSelect)

Returns the internal SQL which defines the specified (related)foundset.
 Table filters are on by default.
 Make sure to set the applicable filters when the sql is used in a loadRecords() call.

Parameters

[Object](#) foundsetOrQBSelect The JSFoundset or QBSelect to get the sql for.

Returns

[String](#) String representing the sql of the JSFoundset.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var sql = databaseManager.getSQL(foundset)
```

getSQL(foundsetOrQBSelect, includeFilters)

Returns the internal SQL which defines the specified (related)foundset.
 Optionally, the foundset and table filter params can be excluded in the sql (includeFilters=false).
 Make sure to set the applicable filters when the sql is used in a loadRecords() call.
 When the founset is in find mode, the find conditions are included in the resulting query.

Parameters

[Object](#) foundsetOrQBSelect The JSFoundset or QBSelect to get the sql for.

[Boolean](#) includeFilters include the foundset and table filters.

Returns

[String](#) String representing the sql of the JSFoundset.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var sql = databaseManager.getSQL(foundset)
```

getSQLParameters(foundsetOrQBSelect)

Returns the internal SQL parameters, as an array, that are used to define the specified (related)foundset.
 Parameters for the filters are included.

Parameters

[Object](#) foundsetOrQBSelect The JSFoundset or QBSelect to get the sql parameters for.

Returns

[Array](#) An Array with the sql parameter values.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var sqlParameterArray = databaseManager.getSQLParameters(foundset,false)
```

getSQLParameters(foundsetOrQBSelect, includeFilters)

Returns the internal SQL parameters, as an array, that are used to define the specified (related)foundset.
 When the founset is in find mode, the arguments for the find conditions are included in the result.

Parameters

Object foundsetOrQBSelect The JSFoundset or QBSelect to get the sql parameters for.
Boolean includeFilters include the parameters for the filters.

Returns

Array An Array with the sql parameter values.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var sqlParameterArray = databaseManager.getSQLParameters(foundset,false)
```

getServerNames()

Returns an array with all the server names used in the solution.

NOTE: For more detail on named server connections, see the chapter on Database Connections, beginning with the Introduction to database connections in the Servoy Developer User's Guide.

Returns

Array An Array of servernames.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var array = databaseManager.getServerNames()
```

getTable(foundset)

Returns the JSTable object from which more info can be obtained (like columns).
 The parameter can be a JSFoundset,JSRecord,datasource string or server/tablename combination.

Parameters

JSFoundSet foundset The foundset where the JSTable can be get from.

Returns

JSTable the JSTable get from the input.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var jstable = databaseManager.getTable(controller.getDataSource());
//var jstable = databaseManager.getTable(foundset);
//var jstable = databaseManager.getTable(record);
//var jstable = databaseManager.getTable(datasource);
var tableSQLName = jstable.getSQLName();
var columnNamesArray = jstable.getColumnNames();
var firstColumnName = columnNamesArray[0];
var jscolumn = jstable.getColumn(firstColumnName);
var columnLength = jscolumn.getLength();
var columnType = jscolumn.getTypeAsString();
var columnSQLName = jscolumn.getSQLName();
var isPrimaryKey = jscolumn.isRowIdentifier();
```

getTable(record)

Returns the JSTable object from which more info can be obtained (like columns).
 The parameter can be a JSFoundset,JSRecord,datasource string or server/tablename combination.

Parameters

JSRecord record The record where the table can be get from.

Returns

JSTable the JSTable get from the input.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

var jstable = databaseManager.getTable(controller.getDataSource());
//var jstable = databaseManager.getTable(foundset);
//var jstable = databaseManager.getTable(record);
//var jstable = databaseManager.getTable(datasource);
var tableSQLName = jstable.getSQLName();
var columnNamesArray = jstable.getColumnNames();
var firstColumnName = columnNamesArray[0];
var jscolumn = jstable.getColumn(firstColumnName);
var columnLength = jscolumn.getLength();
var columnType = jscolumn.getTypeAsString();
var columnSQLName = jscolumn.getSQLName();
var isPrimaryKey = jscolumn.isRowIdentifier();

```

getTable(dataSource)

Returns the JSTable object from which more info can be obtained (like columns).
The parameter can be a JSFoundset, JSRecord, datasource string or server/tablename combination.

Parameters

[String](#) dataSource The datasource where the table can be get from.

Returns

[JSTable](#) the JSTable get from the input.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```

var jstable = databaseManager.getTable(controller.getDataSource());
//var jstable = databaseManager.getTable(foundset);
//var jstable = databaseManager.getTable(record);
//var jstable = databaseManager.getTable(datasource);
var tableSQLName = jstable.getSQLName();
var columnNamesArray = jstable.getColumnNames();
var firstColumnName = columnNamesArray[0];
var jscolumn = jstable.getColumn(firstColumnName);
var columnLength = jscolumn.getLength();
var columnType = jscolumn.getTypeAsString();
var columnSQLName = jscolumn.getSQLName();
var isPrimaryKey = jscolumn.isRowIdentifier();

```

getTable(serverName, tableName)

Returns the JSTable object from which more info can be obtained (like columns).
The parameter can be a JSFoundset, JSRecord, datasource string or server/tablename combination.

Parameters

[String](#) serverName Server name.

[String](#) tableName Table name.

Returns

[JSTable](#) the JSTable get from the input.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```

var jstable = databaseManager.getTable(controller.getDataSource());
//var jstable = databaseManager.getTable(foundset);
//var jstable = databaseManager.getTable(record);
//var jstable = databaseManager.getTable(datasource);
var tableSQLName = jstable.getSQLName();
var columnNamesArray = jstable.getColumnNames();
var firstColumnName = columnNamesArray[0];
var jscolumn = jstable.getColumn(firstColumnName);
var columnLength = jscolumn.getLength();
var columnType = jscolumn.getTypeAsString();
var columnSQLName = jscolumn.getSQLName();
var isPrimaryKey = jscolumn.isRowIdentifier();

```

getTableCount(dataSource)

Returns the total number of records(rows) in a table.

NOTE: This can be an expensive operation (time-wise) if your resultset is large

Parameters

[Object](#) dataSource Data where a server table can be get from. Can be a foundset, a datasource name or a JSTable.

Returns

[Number](#) the total table count.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

//return the total number of rows in a table.
var count = databaseManager.getTableCount(foundset);

```

getTableFilterParams(serverName)

Returns a two dimensional array object containing the table filter information currently applied to the servers tables.

For column-based table filters, a row of 5 fields per filter are returned.

The "columns" of a row from this array are: tablename, dataprovider, operator, value, filtername

For query-based filters, a row of 2 fields per filter are returned.

The "columns" of a row from this array are: query, filtername

Parameters

[String](#) serverName The name of the database server connection.

Returns

[Array](#) Two dimensional array.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

var params = databaseManager.getTableFilterParams(databaseManager.getDataSourceServerName(controller.
getDataSource()))
for (var i = 0; params != null && i < params.length; i++)
{
    if (params[i].length() == 5) {
        application.output('Table filter on table ' + params[i][0] + ': ' + params[i][1] + ' '+params[i]
[2] + ' '+params[i][3] + (params[i][4] == null ? ' [no name]' : ' ['+params[i][4]+'']))
    }
    if (params[i].length() == 2) {
        application.output('Table filter with query ' + params[i][0]+ ': ' + (params[i][1] == null ? '
[no name]' : ' ['+params[i][1]+'']))
    }
}

```

getTableFilterParams(serverName, filterName)

Returns a two dimensional array object containing the table filter information currently applied to the servers tables.

For column-based table filters, a row of 5 fields per filter are returned.

The "columns" of a row from this array are: tablename, dataprovider, operator, value, filtername

For query-based filters, a row of 2 fields per filter are returned.

The "columns" of a row from this array are: query, filtername

Parameters

[String](#) `serverName` The name of the database server connection.

[String](#) `filterName` The filter name for which to get the array.

Returns

[Array](#) Two dimensional array.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
var params = databaseManager.getTableFilterParams(databaseManager.getDataSourceServerName(controller.
getDataSource()))
for (var i = 0; params != null && i < params.length; i++)
{
    if (params[i].length() == 5) {
        application.output('Table filter on table ' + params[i][0] + ': ' + params[i][1] + ' '+params[i]
[2] + ' '+params[i][3] + (params[i][4] == null ? ' [no name]' : ' ['+params[i][4]+'']))
    }
    if (params[i].length() == 2) {
        application.output('Table filter with query ' + params[i][0]+ ': ' + (params[i][1] == null ? '
[no name]' : ' ['+params[i][1]+'']))
    }
}
```

getTableNames(serverName)

Returns an array of all table names for a specified server.

Parameters

[String](#) `serverName` The server name to get the table names from.

Returns

[Array](#) An Array with the tables names of that server.

Supported Clients

SmartClient, WebClient, NGClient

Sample

```
//return all the table names as array
var tableNamesArray = databaseManager.getTableNames('user_data');
var firstTableName = tableNamesArray[0];
```

getViewFoundSet(name)

Returns a ViewFoundSet that was created by getViewFoundSet(name,query,register) with the registered boolean "true".

So it is registered and remembered by the system to use in Forms.

You can't get ViewFounSet back that are not registered to the system, those are not remembered.

Parameters

[String](#) `name` The name to lookup a ViewFoundSet for

Returns

[JSFoundSet](#) A new ViewFoundSet for that query.

Supported Clients

SmartClient, WebClient, NGClient

Sample

getViewFoundSet(name, query)

Returns a foundset object for a specified query.
 This just creates one without keeping any reference to it, you have to use the `getViewFoundSet(name,query,true)` for registering it to the system.
 ViewFoundSets are different then normal foundsets because they have a lot less methods, stuff like `newRecord` /`deleteRecord` don't work.

If you query the pk with the columns that you display for the main or join tables then those columns can be updated and through `ViewFoundSet#save(ViewRecord)` they can be saved.
 If there are changes in ViewRecords of this ViewFoundSet then databroadcast configurations that need to load new data won't do the query right away (only after the save)
 Also loading more (the next chunksize) will not be done. This is because the ViewRecord on an index can be completely changed. We can't track those.

Also databroadcast can be enabled by calling one of the `ViewFoundSet#enableDatabroadcastFor(QBTableClause)` to listen for that specific table (main or joins).
 Flags can be used to control what exactly should be monitored, some don't cost a lot of overhead others have to do a full re-query to see the changes.

Parameters

String name The name given to this foundset (will create a datasource url like `view:[name]`)
QBSelect query The query to get the ViewFoundSet for.

Returns

JSFoundSet A new ViewFoundSet for that query.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// create a new view foundset and also directly register it to the system so they can be picked up by forms if a
// form has the view datasource.
/** @type {ViewFoundSet<view:myname>} */
var vfs = databaseManager.getViewFoundSet('myname', query, true)
```

getViewFoundSet(name, query, register)

Returns a foundset object for a specified query. If the boolean register is true then the system will register it to the system so it can be used in Forms.
 Also `getViewFoundSet(name)` will then return that instance.
 ViewFoundSets are different then normal foundsets because they have a lot less methods, stuff like `newRecord` /`deleteRecord` don't work.

If you query the pk with the columns that you display for the main or join tables then those columns can be updated and through `ViewFoundSet#save(ViewRecord)` they can be saved.
 If there are changes in ViewRecords of this ViewFoundSet then databroadcast configurations that need to load new data won't do the query right away (only after the save)
 Also loading more (the next chunksize) will not be done. This is because the ViewRecord on an index can be completely changed. We can't track those.

Also databroadcast can be enabled by calling one of the `ViewFoundSet#enableDatabroadcastFor(QBTableClause)` to listen for that specific table (main or joins).
 Flags can be used to control what exactly should be monitored, some don't cost a lot of overhead others have to do a full re-query to see the changes.

if the register boolean is true, then the given ViewFoundSet is registered to the system so it is picked up by forms that have this datasource (see `viewFoundset.getDatasource()` for the actual datasource string) assigned.
 The form's foundset will then have a much more limited API, so a lot of things can't be done with it - e.g. `newRecord()` or `deleteRecords()`.
 Also records can be updated in memory, so they are not fully read-only, but the developer is responsible for saving these changes to a persisted store. See also `viewFoundset.save(...)`.

If the solution doesn't need this ViewFoundSet anymore and you did use register is true, please use `ViewFoundSet.dispose()` to clear and remove it from the system, because otherwise this register call will keep/hold this foundset in memory (for that datasource string to work) forever.

Parameters

String name The name given to this foundset (will create a datasource url like `view:[name]`)
QBSelect query The query to get the ViewFoundSet for.
Boolean register Register the created ViewFoundSet to the system so it can be used by forms.

Returns

JSFoundSet A new JSFoundset for that query.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// create a new view foundset and also directly register it to the system so they can be picked up by forms if a
form has the view datasource.
/** @type {ViewFoundSet<view:myname>} */
var vfs = databaseManager.getViewFoundSet('myname', query, true)
```

getViewNames(serverName)

Returns an array of all view names for a specified server.

Parameters

[String](#) serverName The server name to get the view names from.

Returns

[Array](#) An Array with the view names of that server.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//return all the view names as array
var viewNamesArray = databaseManager.getViewNames('user_data');
var firstViewName = viewNamesArray[0];
```

hasLocks()

Returns true if the current client has any or the specified lock(s) acquired.

Returns

[Boolean](#) true if the current client has locks or the lock.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var hasLocks = databaseManager.hasLocks('mylock')
```

hasLocks(lockName)

Returns true if the current client has any or the specified lock(s) acquired.

Parameters

[String](#) lockName The lock name to check.

Returns

[Boolean](#) true if the current client has locks or the lock.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var hasLocks = databaseManager.hasLocks('mylock')
```

hasNewRecords(foundset)

Returns true if the argument (foundSet / record) has at least one row that was not yet saved in the database.

Parameters

[JSFoundSet](#) foundset The JSFoundset to test.

Returns

[Boolean](#) true if the JSFoundset has new records or JSRecord is a new record.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var fs = databaseManager.getFoundSet(databaseManager.getDataSourceServerName(controller.getDataSource()),
'employees');
databaseManager.startTransaction();
var ridx = fs.newRecord();
var record = fs.getRecord(ridx);
record.emp_name = 'John';
if (databaseManager.hasNewRecords(fs)) {
    application.output("new records");
} else {
    application.output("no new records");
}
databaseManager.saveData();
databaseManager.commitTransaction();
```

hasNewRecords(foundset, index)

Returns true if the argument (foundSet / record) has at least one row that was not yet saved in the database.

Parameters

[JSFoundSet](#) foundset The JSFoundset to test.

[Number](#) index The record index in the foundset to test (not specified means has the foundset any new records)

Returns

[Boolean](#) true if the JSFoundset has new records or JSRecord is a new record.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var fs = databaseManager.getFoundSet(databaseManager.getDataSourceServerName(controller.getDataSource()),
'employees');
databaseManager.startTransaction();
var ridx = fs.newRecord();
var record = fs.getRecord(ridx);
record.emp_name = 'John';
if (databaseManager.hasNewRecords(fs)) {
    application.output("new records");
} else {
    application.output("no new records");
}
databaseManager.saveData();
databaseManager.commitTransaction();
```

hasRecordChanges(foundset)

Returns true if the specified foundset, on a specific index or in any of its records, or the specified record has changes or is new unsaved record.

NOTE: The fields focus may be lost in user interface in order to determine the edits.

Parameters

[JSFoundSet](#) foundset The JSFoundset to test if it has changes.

Returns

[Boolean](#) true if there are changes in the JSFoundset or JSRecord.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
if (databaseManager.hasRecordChanges(foundset,2))
{
    //do save or something else
}
```

hasRecordChanges(foundset, index)

Returns true if the specified foundset, on a specific index or in any of its records, or the specified record has changes or is new unsaved record.

NOTE: The fields focus may be lost in user interface in order to determine the edits.

Parameters

[JSFoundSet](#) foundset The JSFoundset to test if it has changes.

[Number](#) index The record index in the foundset to test (not specified means has the foundset any changed records)

Returns

[Boolean](#) true if there are changes in the JSFoundset or JSRecord.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
if (databaseManager.hasRecordChanges(foundset,2))
{
    //do save or something else
}
```

hasRecords(foundset)

Returns true if the (related)foundset exists and has records.

Parameters

[JSFoundSet](#) foundset A JSFoundset to test.

Returns

[Boolean](#) true if the foundset/relation has records.

Supported Clients

SmartClient,WebClient,NGClient,MobileClient

Sample

```
if (%%elementName%%.hasRecords(orders_to_orderitems))
{
    //do work on relatedFoundSet
}
//if (%%elementName%%.hasRecords(foundset.getSelectedRecord(),'orders_to_orderitems.orderitems_to_products'))
//{
//    //do work on deeper relatedFoundSet
//}
```

hasRecords(record, relationString)

Returns true if the (related)foundset exists and has records.

Parameters

[JSRecord](#) record A JSRecord to test.

[String](#) relationString The relation name.

Returns

[Boolean](#) true if the foundset/relation has records.

Supported Clients

SmartClient,WebClient,NGClient,MobileClient

Sample

```
if (%%elementName%%.hasRecords(orders_to_orderitems))
{
    //do work on relatedFoundSet
}
//if (%%elementName%%.hasRecords(foundset.getSelectedRecord(),'orders_to_orderitems.orderitems_to_products'))
//{
//    //do work on deeper relatedFoundSet
//}
```

hasTransaction()

Returns true if there is an transaction active for this client.

Returns

[Boolean](#) true if the client has a transaction.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var hasTransaction = databaseManager.hasTransaction()
```

mergeRecords(sourceRecord, combinedDestinationRecord)

Merge records from the same foundset, updates entire datamodel (via foreign type on columns) with destination record pk, deletes source record. Do use a transaction!

This function is very handy in situations where duplicate data exists. It allows you to merge the two records and move all related records in one go. Say the source_record is "Ikea" and the combined_destination_record is "IKEA", the "Ikea" record is deleted and all records related to it (think of contacts and orders, for instance) will be related to the "IKEA" record.

The function takes an optional array of column names. If provided, the data in the named columns will be copied from source_record to combined_destination_record.

Note that it is essential for both records to originate from the same foundset, as shown in the sample code.

Parameters

[JSRecord](#) sourceRecord The source JSRecord to copy from.
[JSRecord](#) combinedDestinationRecord The target/destination JSRecord to copy into.

Returns

[Boolean](#) true if the records could me merged.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.mergeRecords(foundset.getRecord(1), foundset.getRecord(2));
```

mergeRecords(sourceRecord, combinedDestinationRecord, columnNames)

Merge records from the same foundset, updates entire datamodel (via foreign type on columns) with destination record pk, deletes source record. Do use a transaction!

This function is very handy in situations where duplicate data exists. It allows you to merge the two records and move all related records in one go. Say the source_record is "Ikea" and the combined_destination_record is "IKEA", the "Ikea" record is deleted and all records related to it (think of contacts and orders, for instance) will be related to the "IKEA" record.

The function takes an optional array of column names. If provided, the data in the named columns will be copied from source_record to combined_destination_record.

Note that it is essential for both records to originate from the same foundset, as shown in the sample code.

Parameters

[JSRecord](#) sourceRecord The source JSRecord to copy from.
[JSRecord](#) combinedDestinationRecord The target/destination JSRecord to copy into.
[Array](#) columnNames The column names array that should be copied.

Returns

[Boolean](#) true if the records could me merged.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.mergeRecords(foundset.getRecord(1), foundset.getRecord(2));
```

recalculate(foundsetOrRecord)

Can be used to recalculate a specified record or all rows in the specified foundset.
 May be necessary when data is changed from outside of servoy, or when there is data changed inside servoy but records with calculations depending on that data where not loaded so not updated and you need to update the stored calculation values because you are depending on that with queries or aggregates.

Parameters

[Object](#) foundsetOrRecord JSFoundset or JSRecord to recalculate.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// recalculate one record from a foundset.
databaseManager.recalculate(foundset.getRecord(1));
// recalculate all records from the foundset.
// please use with care, this can be expensive!
//databaseManager.recalculate(foundset);
```

refreshRecordFromDatabase(foundset, index)

Flushes the client data cache and requeries the data for a record (based on the record index) in a foundset or all records in the foundset.
 Used where a program external to Servoy has modified the database record.
 Giving 0 as the index will just refresh the selected record.

If the index is -1 then this method will refresh all the records of the datasource of the foundset, it does this by flushing all the records and the row data of the full datasource
 So everything is reloaded fully fresh when the foundsets will requery for there data.
 WARNING: Don't hold any references to JSRecord objects from before this call with -1 index. Those records objects are all in an invalid state because of the underlying data flush.

Parameters

[Object](#) foundset The JSFoundset to refresh

[Number](#) index The index of the JSRecord that must be refreshed (or -1 for all).

Returns

[Boolean](#) true if the refresh was done.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//refresh the second record from the foundset.
databaseManager.refreshRecordFromDatabase(foundset,2)
//flushes all records in the related foundset datasource (so the whole table, so -1 is an expensive operation)
databaseManager.refreshRecordFromDatabase(foundset,-1);
```

releaseAllLocks()

Release all current locks the client has (optionally limited to named locks).
 return true if the locks are released.

Returns

[Boolean](#) true if all locks or the lock is released.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.releaseAllLocks('mylock')
```

releaseAllLocks(lockName)

Release all current locks the client has (optionally limited to named locks).
 return true if the locks are released.

Parameters

[String](#) lockName The lock name to release.

Returns

[Boolean](#) true if all locks or the lock is released.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.releaseAllLocks('mylock')
```

removeDataSource(uri)

Free resources allocated for a previously created data source.

NOTE: make sure this datasource is not using anymore in forms or components!
because the inmemory table and the foundset build on that table are all just removed.

Normally this will be automatically done if a client is removed/shutdown, but if constantly new stuff is created or you don't need it anymore from what the client currently is using or seeing, then removing this will clean up memory.

Parameters

[String](#) uri;

Returns

[Boolean](#)

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.removeDataSource(uri);
```

removeTableFilterParam(serverName, filterName)

Removes a previously defined table filter.

Parameters

[String](#) serverName The name of the database server connection.

[String](#) filterName The name of the filter that should be removed.

Returns

[Boolean](#) true if the filter could be removed.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var success = databaseManager.removeTableFilterParam('admin', 'highNumberedMessagesRule')
```

revertEditedRecords()

Reverts outstanding (not saved) in memory changes from edited records.

Can specify a record or foundset as parameter to rollback.

Best used in combination with the function databaseManager.setAutoSave()

This does not include deletes, they do not honor the autosave false flag so they cant be rollbacked by this call.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//Set autosave, if false then no saves will happen by the ui (not including deletes!). Until you call saveData
or setAutoSave(true)
//reverts in mem the records that were edited and not yet saved. Best used in combination with autosave false.
databaseManager.setAutoSave(false)
//Now let users input data

//On save or cancel, when data has been entered:
if (cancel) databaseManager.revertEditedRecords()
//databaseManager.revertEditedRecords(foundset); // rollback all records from foundset
//databaseManager.revertEditedRecords(foundset.getSelectedRecord()); // rollback only one record
databaseManager.setAutoSave(true)
```

revertEditedRecords(foundset)

Reverts outstanding (not saved) in memory changes from edited records.

Can specify a record or foundset as parameter to rollback.

Best used in combination with the function `databaseManager.setAutoSave()`

This does not include deletes, they do not honor the autosave false flag so they cant be rolledback by this call.

Parameters

[JSFoundSet](#) foundset A JSFoundset to revert.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//Set autosave, if false then no saves will happen by the ui (not including deletes!). Until you call saveData
or setAutoSave(true)
//reverts in mem the records that were edited and not yet saved. Best used in combination with autosave false.
databaseManager.setAutoSave(false)
//Now let users input data

//On save or cancel, when data has been entered:
if (cancel) databaseManager.revertEditedRecords()
//databaseManager.revertEditedRecords(foundset); // rollback all records from foundset
//databaseManager.revertEditedRecords(foundset.getSelectedRecord()); // rollback only one record
databaseManager.setAutoSave(true)
```

rollbackTransaction()

Rollback a transaction started by `databaseManager.startTransaction()`.

Note that when autosave is false, `revertEditedRecords()` will not handle deleted records, while `rollbackTransaction()` does.

Also, `rollbackEditedRecords()` is called before rolling back the transaction see `rollbackTransaction(boolean)` to control that behavior

and saved records within the transactions are restored to the database values, so user input is lost, to control this see `rollbackTransaction(boolean,boolean)`

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceled or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
    databaseManager.rollbackTransaction();
}
```

rollbackTransaction(rollbackEdited)

Rollback a transaction started by `databaseManager.startTransaction()`.

Note that when autosave is false, `revertEditedRecords()` will not handle deleted records, while `rollbackTransaction()` does.

Also, saved records within the transactions are restored to the database values, so user input is lost, to controll this see `rollbackTransaction(boolean,boolean)`

Parameters

[Boolean](#) rollbackEdited call `rollbackEditedRecords()` before rolling back the transaction

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceled or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
    databaseManager.rollbackTransaction();
}
```

rollbackTransaction(rollbackEdited, revertSavedRecords)

Rollback a transaction started by databaseManager.startTransaction().

Note that when autosave is false, revertEditedRecords() will not handle deleted records, while rollbackTransaction() does.

Parameters

Bo rollbackE call rollbackEditedRecords() before rolling back the transaction
ole dited
an
Bo revertSav if false then all records in the transaction do keep the user input and are back in the edited records list. Note that if the pks of such a
ole edRecordsrecord are no longer used by it's foundset (find/search or load by query or ...) it will just be rolled-back as it can't be put in editing records
an list.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceled or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
    databaseManager.rollbackTransaction();
}
```

saveData()

Saves all outstanding (unsaved) data and exits the current record.

Optionally, by specifying a record or foundset, can save a single record or all records from foundset instead of all the data.

Since Servoy 8.3 saveData with null parameter does not call saveData() as fallback, it just returns false.

NOTE: The fields focus may be lost in user interface in order to determine the edits.

SaveData called from table events (like afterRecordInsert) is only partially supported depending on how first saveData (that triggers the event) is called.

If first saveData is called with no arguments, all saveData from table events are returning immediately with true value and records will be saved as part of first save.

If first saveData is called with record(s) as arguments, saveData from table event will try to save record(s) from arguments that are different than those in first call.

SaveData with no arguments inside table events will always return true without saving anything.

Returns

Boolean true if the save was done without an error.

Supported Clients

SmartClient,WebClient,NGClient,MobileClient

Sample

```

databaseManager.saveData();
//databaseManager.saveData(foundset.getRecord(1));//save specific record
//databaseManager.saveData(foundset);//save all records from foundset

// when creating many records in a loop do a batch save on an interval as every 10 records (to save on memory
and roundtrips)
// for (var recordIndex = 1; recordIndex <= 5000; recordIndex++)
// {
//     foundset.newRecord();
//     someColumn = recordIndex;
//     anotherColumn = "Index is: " + recordIndex;
//     if (recordIndex % 10 == 0) databaseManager.saveData();
// }

```

saveData(foundset)

Saves all outstanding (unsaved) data and exits the current record. Optionally, by specifying a record or foundset, can save a single record or all records from foundset instead of all the data.

Since Servoy 8.3 saveData with null parameter does not call saveData() as fallback, it just returns false.

NOTE: The fields focus may be lost in user interface in order to determine the edits.

SaveData called from table events (like afterRecordInsert) is only partially supported depending on how first saveData (that triggers the event) is called.

If first saveData is called with no arguments, all saveData from table events are returning immediately with true value and records will be saved as part of first save.

If first saveData is called with record(s) as arguments, saveData from table event will try to save record(s) from arguments that are different than those in first call.

SaveData with no arguments inside table events will always return true without saving anything.

Parameters

[JSFoundSet](#) foundset The JSFoundset to save.

Returns

[Boolean](#) true if the save was done without an error.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```

databaseManager.saveData();
//databaseManager.saveData(foundset.getRecord(1));//save specific record
//databaseManager.saveData(foundset);//save all records from foundset

// when creating many records in a loop do a batch save on an interval as every 10 records (to save on memory
and roundtrips)
// for (var recordIndex = 1; recordIndex <= 5000; recordIndex++)
// {
//     foundset.newRecord();
//     someColumn = recordIndex;
//     anotherColumn = "Index is: " + recordIndex;
//     if (recordIndex % 10 == 0) databaseManager.saveData();
// }

```

saveData(record)

Saves all outstanding (unsaved) data and exits the current record.

Optionally, by specifying a record or foundset, can save a single record or all records from foundset instead of all the data.

Since Servoy 8.3 saveData with null parameter does not call saveData() as fallback, it just returns false.

NOTE: The fields focus may be lost in user interface in order to determine the edits.

SaveData called from table events (like afterRecordInsert) is only partially supported depending on how first saveData (that triggers the event) is called.

If first saveData is called with no arguments, all saveData from table events are returning immediately with true value and records will be saved as part of first save.

If first saveData is called with record(s) as arguments, saveData from table event will try to save record(s) from arguments that are different than those in first call.

SaveData with no arguments inside table events will always return true without saving anything.

Parameters

[JSRecord](#) record The JSRecord to save.

Returns

[Boolean](#) true if the save was done without an error.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.saveData();
//databaseManager.saveData(foundset.getRecord(1));//save specific record
//databaseManager.saveData(foundset);//save all records from foundset

// when creating many records in a loop do a batch save on an interval as every 10 records (to save on memory
and roundtrips)
// for (var recordIndex = 1; recordIndex <= 5000; recordIndex++)
// {
//     foundset.newRecord();
//     someColumn = recordIndex;
//     anotherColumn = "Index is: " + recordIndex;
//     if (recordIndex % 10 == 0) databaseManager.saveData();
// }
```

saveData(records)

Saves all outstanding (unsaved) data and exits the current record.
Optionally, by specifying a record or foundset, can save a single record or all records from foundset instead of all the data.

Since Servoy 8.3 saveData with null parameter does not call saveData() as fallback, it just returns false.

NOTE: The fields focus may be lost in user interface in order to determine the edits.

SaveData called from table events (like afterRecordInsert) is only partially supported depending on how first saveData (that triggers the event) is called.

If first saveData is called with no arguments, all saveData from table events are returning immediately with true value and records will be saved as part of first save.

If first saveData is called with record(s) as arguments, saveData from table event will try to save record(s) from arguments that are different than those in first call.

SaveData with no arguments inside table events will always return true without saving anything.

Parameters

[Array](#) records The array of JSRecord to save.

Returns

[Boolean](#) true if the save was done without an error.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.saveData();
//databaseManager.saveData(foundset.getRecord(1));//save specific record
//databaseManager.saveData(foundset);//save all records from foundset

// when creating many records in a loop do a batch save on an interval as every 10 records (to save on memory
and roundtrips)
// for (var recordIndex = 1; recordIndex <= 5000; recordIndex++)
// {
//     foundset.newRecord();
//     someColumn = recordIndex;
//     anotherColumn = "Index is: " + recordIndex;
//     if (recordIndex % 10 == 0) databaseManager.saveData();
// }
```

setAutoSave(autoSave)

Set autosave, if false then no saves will happen by the ui (not including deletes!).
Until you call databaseManager.saveData() or setAutoSave(true)

If you also want to be able to rollback deletes then you have to use databaseManager.startTransaction().
Because even if autosave is false deletes of records will be done.

Parameters

[Boolean](#) autoSave Boolean to enable or disable autosave.

Returns

[Boolean](#) false if the current edited record could not be saved.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//Rollbacks in mem the records that were edited and not yet saved. Best used in combination with autosave false.
databaseManager.setAutoSave(false)
//Now let users input data

//On save or cancel, when data has been entered:
if (cancel) databaseManager.rollbackEditedRecords()
databaseManager.setAutoSave(true)
```

setCreateEmptyFormFoundsets()

Turnoff the initial form foundset record loading, set this in the solution open method. Similar to calling foundset.clear() in the form's onload event.

NOTE: When the foundset record loading is turned off, controller.find or controller.loadAllRecords must be called to display the records

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//this has to be called in the solution open method
databaseManager.setCreateEmptyFormFoundsets()
```

startTransaction()

Start a database transaction.

If you want to avoid round trips to the server or avoid the possibility of blocking other clients because of your pending changes, you can use databaseManager.setAutoSave(false/true) and databaseManager.rollbackEditedRecords().

startTransaction, commit/rollbackTransacton() does support rollbacking of record deletes which autoSave = false doesnt support.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceld or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
    databaseManager.rollbackTransaction();
}
```

switchServer(sourceName, destinationName)

Switches a named server to another named server with the same datamodel (recommended to be used in an onOpen method for a solution). return true if successful.

Note that this only works if source and destination server are of the same database type.

Parameters

[String](#) sourceName The name of the source database server connection

[String](#) destinationName The name of the destination database server connection.

Returns

[Boolean](#) true if the switch could be done.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
//dynamically changes a server for the entire solution, destination database server must contain the same tables
//columns!
//will fail if there is a lock, transaction , if repository_server is used or if destination server is invalid
//in the solution keep using the sourceName every where to reference the server!
var success = databaseManager.switchServer('crm', 'crm1')
```

updateTableFilterParam(serverName, filterName, query)

Updates a previously defined table filter. Server/Table should not be changed.

Parameters

String serverName The name of the database server connection.
String filterName The name of the filter that should be updated.
QBSelect query condition to filter on.

Returns

Boolean true if the filter could be updated.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
var success = databaseManager.updateTableFilterParam('admin', 'highNumberedMessagesRule', query)
```

updateTableFilterParam(serverName, filterName, tableName, dataprovider, operator, value)

Updates a filter with a new condition. The server/table name should be unchanged.

Parameters

String serverName The name of the database server connection for the specified table name.
String filterName The name of the filter that should be updated.
String tableName The name of the specified table.
String dataprov A specified dataprovider column name.
String operator One of "=", "<", ">", ">=", "<=", "!=", "LIKE", or "IN" optionally augmented with modifiers "#" (ignore case) or "^|" (or-is-null), prefix with "sql:" to allow the value to be interpreted as a custom query.
Object value The specified filter value.

Returns

Boolean true if the tablefilter could be updated.

Supported Clients

SmartClient,WebClient,NGClient

Sample

```
databaseManager.updateTableFilterParam('database', 'myfilter', 'your_i18n_table', 'message_variant', 'in', [1, 2])
```

validate(record)

Validates the given record, it runs first the method that is attached to the entity event "onValidate". Then it will call also the entity events "onInsert" or "onUpdate" depending if the record is new or an update. All those methods do get a parameter JSRecordMarkers where the problems can be reported against.

All columns are then also null/empty checked and if they are and the Column is marked as "not null" an error will be added with the message key "servoy.record.error.null.not.allowed" for that column.

All changed columns are length checked and if the record values is bigger then what the database column can handle and error will be added with the message key "servoy.record.error.columnSizeTooSmall" for that column. Then all the column validators will be run over all the changed columns, The validators will also get the same JSRecordMarkers to report problems to. So the global method validator now also has more parameters then just the value.

These 3 validations (null, length and column validators) are not by default done any more on change of the dataprovider itself. This is controlled by the servoy property "servoy.execute.column.validators.only.on.validate_and_save" which can also be seen at the TableEditor column validators page.

An extra state object can be given that will also be passed around if you want to have more state in the validation objects (like giving some ui state so the entity methods know where you come from)

It will return a JSRecordMarkers when the record had validation problems

Parameters

[JSRecord](#) record The record to validate.

Returns

[JSRecordMarkers](#) Returns a JSRecordMarkers if the record has validation problems

Supported Clients

SmartClient,WebClient,NGClient

Sample

validate(record, customObject)

Validates the given record, it runs first the method that is attached to the entity event "onValidate". Then it will call also the entity events "onInsert" or "onUpdate" depending if the record is new or an update. All those methods do get a parameter JSRecordMarkers where the problems can be reported against.

All columns are then also null/empty checked and if they are and the Column is marked as "not null" an error will be added with the message key "servoy.record.error.null.not.allowed" for that column.

All changed columns are length checked and if the record values is bigger then what the database column can handle and error will be added with the message key "servoy.record.error.columnSizeTooSmall" for that column. Then all the column validators will be run over all the changed columns, The validators will also get the same JSRecordMarkers to report problems to. So the global method validator now also has more parameters then just the value.

These 3 validations (null, length and column validators) are not by default done any more on change of the dataprovider itself. This is controlled by the servoy property "servoy.execute.column.validators.only.on.validate_and_save" which can also be seen at the TableEditor column validators page.

An extra state object can be given that will also be passed around if you want to have more state in the validation objects (like giving some ui state so the entity methods know where you come from)

It will return a JSRecordMarkers when the record had validation problems

Parameters

[JSRecord](#) record The record to validate.
[Object](#) customObject The extra customObject that is passed on the the validation methods.

Returns

[JSRecordMarkers](#)

Supported Clients

SmartClient,WebClient,NGClient

Sample